Dynamic Currency Determination in Optimized Programs

D. M. DHAMDHERE
and
K. V. SANKARANARAYANAN
Indian Institute of Technology, Bombay

Compiler optimizations pose many problems to source level debugging of an optimized program due to reordering, insertion and deletion of code. One such problem is to determine whether the value of a variable is *current* at a breakpoint—that is, whether its actual value is the same as its expected value. We use the notion of *dynamic currency* of a variable in source level debugging and propose the use of a *minimal unrolled graph* to reduce the run-time overhead of dynamic currency determination. We prove that the minimal unrolled graph is an adequate basis for performing bit-vector data flow analyses at a breakpoint. This property is used to perform dynamic currency determination. It is also shown to help in recovery of a dynamically noncurrent variable.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids; symbolic execution; tracing*; D.3.4 [**Software Engineering**]: Programming Languages—*compilers; debuggers; Run-time environments*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: dynamic currency determination, minimal unrolled graph, debugging optimized code, dynamic slicing, compiler, code optimization, source level debugging, code instrumentation

1. INTRODUCTION

A source level debugger is used to examine (and possibly control) the state of a program during its execution. A user specifies a set of *breakpoints* for the purpose of debugging. Each breakpoint is a source statement; the debugger opens a debugging conversation with the user when execution reaches a breakpoint. Compiler optimizations hinder many functionalities of a source level debugger due to reordering, insertion, and deletion of code. This leads to two kinds of problems—code location problems and data value problems.

The *code location problem* is to find a mapping between statements in a source program and instructions in an object program. Such a mapping is needed to set

Author to whom correspondence should be addressed: D. M. Dhamdhere, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, Mumbai 400 076 India. *E-Mail*: `dmd@cse.iitb.ernet.in`.
K. V. Sankaranarayanan was supported by Tandem Corporation, Inc.

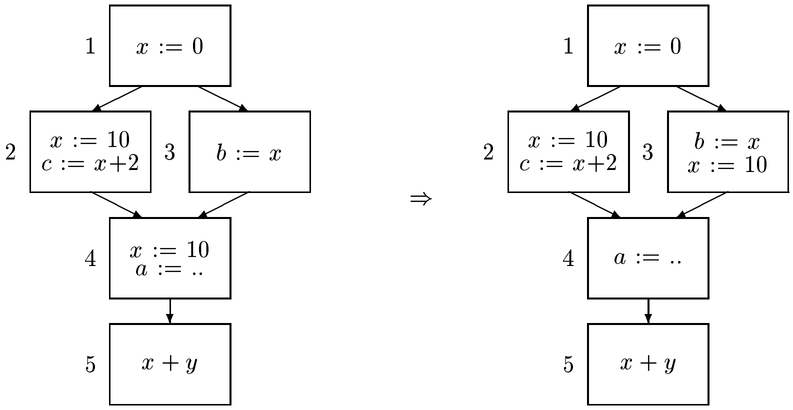

Fig. 1. Insertion and deletion of computations during optimization

breakpoints, while a converse mapping is needed to report the location of run-time exceptions in terms of source program statements. These mappings constitute the *breakpoint model*. Reordering, insertion and deletion of code lead to obvious difficulties in developing the breakpoint model.

An *expected value* of a variable is a value the user expects the variable to have at a breakpoint. The *actual value* of a variable is the value in the location allocated to the variable at a breakpoint. A variable is said to be *current* at a breakpoint if the expected and actual values at the breakpoint are the same in every execution of the program. A variable is said to be *noncurrent* at a breakpoint if its expected and actual values at the breakpoint are different in every execution of the program and *suspect* if the values are the same in some executions and different in other executions. The *data value problem* consists of determining, at a breakpoint, whether the value of a variable exists in a memory location or a register (the *data location problem*) and whether the variable is current, noncurrent or suspect (the *currency determination problem*). Currency determination is vitally important in dynamic debugging of a program because it would be incorrect for a debugger to report the actual value of a noncurrent or suspect variable to the user. *Recovery* is the procedure for obtaining the expected value of a suspect or noncurrent variable at a breakpoint.

Figure 1 shows the program flow graph [Aho et al. 1986] of a program before and after optimization. The program has been optimized by moving the assignment $x := 10$ from node 4 to node 3 to suppress its redundancy along path 1-2-4 in the program. Consider a breakpoint set at the start of node 4. The actual value of x at the breakpoint is the value found in its memory location. This value is '10'. The expected value would depend on the path followed during execution; it would be '10' if execution followed the path 1-2-4, and '0' if execution followed the path 1-3-4. Thus, the value of x is suspect at the start of node 4. Note that x is current at the start of node 5. This is essential since node 5 contains a use of x in the optimized program (else the optimized program would be incorrect!).

Currency determination involves analysis of all possible paths in the original and optimized programs to determine whether the expected and actual values of a variable would be identical at a breakpoint. This analysis can be performed using standard methods of data flow analysis [Hecht 1977; Aho et al. 1986]. Hence this approach is called *static currency determination*. In Figure 1 static currency determination would deduce that x is suspect due to the reasons mentioned earlier. Static currency determination is safe in that it considers all possible paths in a program and declares a variable to be current only if its expected and actual values are the same along all paths. However, the information provided by static currency determination is not precise enough for the purpose of dynamic debugging. For example, in Figure 1 the value of x can be reported at the breakpoint if execution had followed path 1-2-4, however static currency determination forbids this by declaring x to be suspect.

We use the notion of *dynamic currency* of a variable [Copperman 1993; 1994] to indicate whether the actual and expected values of a variable are the same at a breakpoint in a specific execution of a program. This provides a precise answer to the currency determination problem. As is obvious, run-time information is used to determine the dynamic currency of a variable. We use the term *dynamic currency determination* for such an approach.

This paper describes a method for dynamic currency determination in a framework analogous to the static currency determination framework reported in [Adl-Tabatabai 1996]. Our method uses a representation of an execution of a program for the purpose of data flow analysis. This representation is based on run-time information collected through instrumentation of the object code of an optimized program. We use the theory of bit-vector data flow frameworks presented in [Dhamdhere et al. 1992; Khedker and Dhamdhere 1994] to reduce run-time overheads and to obtain an optimized representation called a *minimal unrolled graph*. Dynamic currency determination is performed by data flow analysis over a minimal unrolled graph. We also discuss other interesting applications of a minimal unrolled graph and its limitations resulting from the use of bit-vector data flow frameworks.

Terminology used in currency determination is discussed in Section 2. Section 3 briefly describes related work, mainly Adl-Tabatabai's approach for currency determination using static analysis techniques. In Section 4, we introduce the notion of an unrolled graph to represent the path traversed during an execution of a program and define a minimal unrolled graph as an optimized form of an unrolled graph. Section 5 uses the theory of bit-vector data flow frameworks presented in [Dhamdhere et al. 1992; Khedker and Dhamdhere 1994] to prove an important property of a minimal unrolled graph—at a breakpoint, a bit-vector data flow analysis performed on a minimal unrolled graph yields the same data flow information as a data flow analysis performed on the corresponding unrolled graph. This property is used to perform dynamic currency determination. Section 6 describes how code instrumentation can be used to build a minimal unrolled graph and shows how minimal unrolled graphs can be used for other interesting applications. Recovery of dynamically noncurrent variables using minimal unrolled graphs is described in detail.

2. TERMINOLOGY

A *program flow graph* (PFG) is a directed graph $G = (N, E, start)$, where N is the set of nodes, each node being a basic block in the program, E is the set of control flow edges and $start \in N$ is the entry node of the graph. A *point* exists either between two adjacent operations op_j and op_{j+1} in a node, before the first operation in a node, or after the last operation in a node. An operation op_j (if any) immediately preceding a point p_i in a node is said to *occur at* point p_i. An *operation* may be a statement in the case of a source flow graph, an IR operation in the case of an intermediate representation, or an instruction in the case of an object flow graph. A *path* is a sequence of points $(p_1, \ldots, p_n)$ such that for each pair $(p_i, p_{i+1})_{1 \le i < n}$, control can immediately reach p_{i+1} from p_i. Alternatively, a path can be specified as a sequence of operations or a sequence of nodes in PFG.

A *breakpoint* is a pair (b_s, b_o) where b_s is a point in the source flow graph at which the user expects the program to break and b_o is the point in the object flow graph where the program actually breaks. The mapping between breakpoints in the source and object programs is termed the *breakpoint model.* An *execution path* is a path $(start, \ldots, O)$ followed during an execution of a program. An *execution path pair* (or simply a *path pair*) is a pair (p_s, p_o) where p_s and p_o are the execution paths in the source and object flow graphs to a breakpoint (b_s, b_o), respectively.

Source variables are mapped to registers and memory locations in an object program. (We use the term *location* to refer to a memory location or a register.) Variables mapped to registers may or may not be assigned home locations in memory. Also locations may be shared between different source variables and a variable may exist in different locations at different points in a program. Due to this, a many-to-many mapping arises between source variables and locations. In a path pair (p_s, p_o), an instruction I in p_o is said to be a *definition of a variable V to L* if I corresponds to an assignment of V in p_s and assigns to a location L. V is *resident in L via a path $p_o = (start, \ldots, O)$* if there exists a definition of V to L reaching O along p_o. V is *resident at a point O* if there exists a location L such that V is resident in L via all paths $(start, \ldots, O)$. V is *nonresident* at O if it is not resident at O—that is, no location is allocated to V—or if different locations are allocated to V along different paths reaching O. A variable is said to possess an *actual* value only if it is resident.

Consider a path pair $(p_s, p_o) \equiv ((start_s, \ldots, b_s), (start_o, \ldots, b_o))$ leading to a breakpoint (b_s, b_o). Currency of a variable V via (p_s, p_o) is defined only if V has an expected value via p_s (i.e., V is not uninitialized along p_s) and V has an actual value via p_o (i.e., V is resident in some location L via p_o). Variable V is current in a location L via path pair (p_s, p_o) iff there exists a source assignment d_s on p_s which assigns to V and reaches the end of p_s, and there exists an object assignment d_o on p_o which corresponds to d_s, assigns to L and reaches the end of p_o. Variable V is noncurrent in L via path pair (p_s, p_o) iff there exists a source assignment d_s on p_s which assigns to V and reaches the end of p_s, and there exists an object assignment d'_o of V on p_o which does not correspond to d_s, assigns to L and reaches the end of p_o.

A variable V is *current* in L at a breakpoint (b_s, b_o) iff it is current in L via every path pair leading to (b_s, b_o) and *noncurrent* if it is noncurrent along every path pair reaching (b_s, b_o). A variable which is neither current nor noncurrent is *suspect*. Variable V is said to be *dynamically current* in L at the breakpoint (b_s, b_o) via path pair (p_s, p_o) if and only if breakpoint (b_s, b_o) is reached by traversing the path pair (p_s, p_o) and V is current in L via path pair (p_s, p_o). A suspect variable may be dynamically current or dynamically noncurrent in a particular execution.

3. RELATED WORK

Hennessy [Hennessy 1982] introduces the notion of currency of a variable. Zellweger [Zellweger 1983] inserts path determiners to collect selective execution history information for debugging programs subjected to control flow optimization. However, she uses this information only to perform many-to-one and one-to-many mappings between source locations and object locations. Copperman [Copperman 1993; 1994], Wismüller [Wismüller 1994], and Adl-Tabatabai [Adl-Tabatabai and Gross 1993b; 1993a; 1996; Adl-Tabatabai 1996] provide solutions to the currency determination problem using static analysis. Their schemes differ in the breakpoint model and in the range and modeling of optimizing transformations. None of them, however, deals with dynamic currency determination. Wismüller's scheme can use runtime information, but details are not reported in literature. His algorithms are complex due to the choice of the breakpoint and optimization models.

[Adl-Tabatabai 1996] terms a noncurrent or suspect variable as endangered, and uses data flow analysis techniques to detect endangered variables. His breakpoint model requires the points b_s and b_o to exist in corresponding source and object nodes. Under this breakpoint model, data value problems caused by global optimizations can be tackled by modeling the optimizations in terms of two core transformations—code hoisting and dead code elimination—and performing static analyses on an intermediate representation (IR).

The data value problem is divided into two components—local and global. The local component is solved by algorithms which handle local optimizations and local instruction scheduling [Adl-Tabatabai and Gross 1993a] and the global component is solved by static analyses. Preconditions for the currency of a variable are tested in two steps: First, reaching definitions analysis is performed on the source flow graph to determine whether the variable has an expected value at the breakpoint. Second, available residences analysis [Adl-Tabatabai and Gross 1993b] is performed on an instruction level IR containing debugging information to determine whether the variable has an actual value at the breakpoint. If these conditions are satisfied, static analyses are performed to determine whether a variable is current. All static analyses performed are bit-vector analyses [Khedker and Dhamdhere 1994].

Code movement is performed in two ways—hoisting and sinking. All through this paper, we discuss code hoisting as a representative transformation for code movement. It is implicit that code sinking can be handled analogously. Hoisting of an expression is modeled as copying an expression E from a node N_i to one or more nodes post-dominated by N_i. Code sinking is modeled as movement of an expression E from a node N_i to one or more nodes dominated by N_i. These assumptions limit the region of code where a variable is endangered and simplify the currency determination algorithms [Adl-Tabatabai 1996]. If an assignment expression E is hoisted, copies of E are inserted at a set of program points. We denote these expressions by E_h. The original expression, which is now redundant and can be

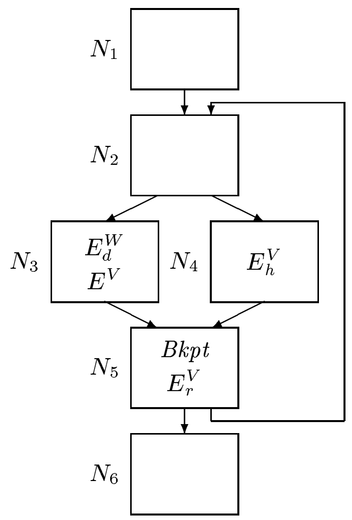

Fig. 2. Representing hoisted and redundant expressions

removed, is called the redundant copy and denoted by E_r. An eliminated dead assignment expression E is denoted by E_d. It is assumed that a compiler introduces annotations in a program flow graph (or in an IR containing sufficient information to build a program flow graph) to indicate these insertions and deletions of code. Figure 2 shows the representation of a program using this model. Optimization has hoisted the assignment expression E^V which assigns to V, from node $N5$ to node $N4$. The inserted expression is denoted by E_h^V and the redundant expression, which is eliminated from the program, is denoted by E_r^V. Note that the flow graph of Figure 2 represents the source program if we assume E_d^W and E_r^V to exist in the program and ignore the presence of E_h^V, and represents the target program if we ignore the presence of E_d^W and E_r^V. Thus, this program representation can be used to determine the currency of E and V using the definitions of Section 2.

In this model, a simpler definition of the currency of a variable is as follows: A redundant assignment expression E^V *hoist-reaches via a path* $P = (start, \ \ldots, \ O)$, if E_h^V reaches O along P, and E_r^V does not occur after the last occurrence of E_h^V along P. V is noncurrent at O if E^V hoist-reaches along all paths $(start, \ldots, O)$ and current if it does not hoist-reach along any path. V is suspect at O if it hoist-reaches via at least one path, but not along all paths, leading to O.

4. THE MINIMAL UNROLLED GRAPH

Following Section 3, dynamic currency of a variable can be defined as follows: V is dynamically current at a breakpoint B via path $P = (start, \ldots, B)$ if E^V does not hoist-reach B via P. To perform dynamic currency determination at a breakpoint, a debugger must collect execution history information to deduce the path followed to the breakpoint. For example, in Figure 2 dynamic currency determination of V at *Bkpt* requires knowledge of which branch was taken at N_2 in the current iteration. An *execution path* is the locus of control during an execution of a program. Thus, an execution path is a (possibly cyclic) path in the PFG of a program; it contains

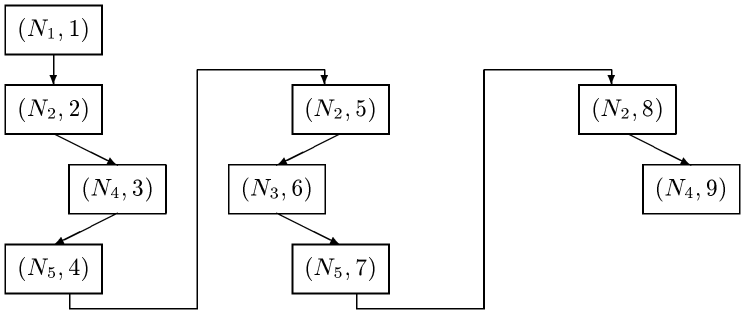

Fig. 3. An unrolled graph

the complete history of control flow during an execution. When we consider the computations occurring at various points in an execution path, we see that an execution path is a representation of an execution of a program. In this section we develop an efficient representation of an execution path.

We associate a time-stamp with a node every time the node (i.e. the code corresponding to it) is executed. A time-stamp can be an integer value obtained from a logical clock—a counter which is incremented every time a node is executed[1]. Note that a node may be executed several times during an execution of a program; a new time-stamp is associated with it at every execution. Thus a pair (N_i, t_j) where N_i is a node and t_j is a time-stamp represents an execution of node N_i. We call each such pair a *block*. We represent an execution path by an unrolled graph.

Definition 4.1. An *unrolled graph* with respect to a program flow graph $G = (N, E, \mathit{start})$ is a graph $G_u = (N_u, E_u, s_u)$ where

(1) N_u is a set of blocks $\{(N_i, t_j)\}$ such that $N_i \in N$ and $t_j > 0$.
(2) $((N_i, t_j), (N_k, t_l)) \in E_u$ iff $(N_i, N_k) \in E$ and $t_l = t_j + 1$.
(3) $s_u = (\mathit{start}, 1)$.

As an example, consider an execution of the program of Figure 2 in which the branches (N_2, N_4), (N_2, N_3) and (N_2, N_4) are taken in the first three iterations of an execution. Figure 3 shows the unrolled graph when execution reaches the breakpoint at the start of node $N5$ in the third iteration. We define a function $\textit{t-stamp}(N_i)$ to return the latest time-stamp of node N_i in an unrolled graph. Here $\textit{t-stamp}(N_2) = 8$, $\textit{t-stamp}(N_4) = 9$ and $\textit{t-stamp}(N_5) = 7$. Figure 4 shows the time stamps of the nodes at the breakpoint. Integers in parentheses indicate time-stamps associated with nodes. It can be seen that the time-stamp of a node N_i is $\textit{t-stamp}(N_i)$.

We can represent the unrolled graph by a list of blocks arranged in ascending order by their time-stamps. We call such a list of blocks an *execution path representation* (EPR). EPR can be constructed during an execution of a program by appending a block (N_i, t_j) to the end of the list when time-stamp t_j is associated with node N_i during the execution of the program. However, this approach suffers

[1]The initial time-stamp of each node is assumed to be 0

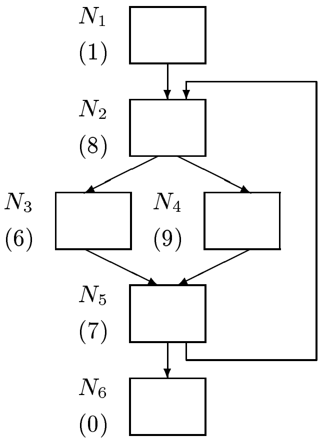

Fig. 4. Time stamps at a breakpoint

from two problems. First, it incurs the run-time overhead of constructing EPR in addition to the overhead of time-stamping. Second, the storage requirements of EPR are unbounded. In the following, we propose an alternative which overcomes these problems.

Definition 4.2. The *minimal unrolled graph* with respect to an unrolled graph G_u is $G_u^m = (N_u^m, E_u^m, s_u^m)$ where $N_u^m = N_u - RN$, $E_u^m = (E_u - RE) \cup AE$ and $s_u^m = s_u$ where

(1) $(N_i, t_j) \in RN$ if $(N_i, t_j) \in N_u$ and $\exists (N_i, t_k) \in N_u$ such that $t_k > t_j$.
(2) $((N_i, t_j), (N_k, t_l)) \in RE$ if $((N_i, t_j), (N_k, t_l)) \in E_u$ and either $(N_i, t_j) \in RN$ or $(N_k, t_l) \in RN$.
(3) $((N_i, t_j), (N_k, t_l)) \in AE$ if $(N_i, t_j), (N_k, t_l) \in N_u^m$, $t_l \geq t_j + 1$ and $\nexists\ (N_g, t_h) \in N_u^m$ such that $t_j < t_h < t_l$.

Thus each (N_i, t_j) in G_u^m represents the last (or only) instance of execution of N_i, i.e. $t_j = \textit{t-stamp}(N_i)$. It is also clear that G_u^m contains a single acyclic path in which the blocks (N_i, t_j) appear in ascending order by time-stamp. Figure 5 shows the minimal unrolled graph for the unrolled graph of Figure 3.

The *minimal execution path representation*(MEPR) is now a list of blocks in G_u^m arranged in ascending order by their time-stamps. Note two important differences between EPR and MEPR. First, a node in G appears at most once in the blocks of MEPR. Second, since $t_j = \textit{t-stamp}(N_i)$ for each block (N_i, t_j) in G_u^m, MEPR can be constructed at a breakpoint. Thus, run-time overhead is restricted to the time overhead of time-stamping and the space overhead of an array of time-stamps whose size is bounded by the number of nodes in G.

We make two observations concerning time-stamps associated with nodes in G_u^m.

Observation 4.1. Each node $N_i \in N$, such that N_i has been executed at least once and $\textit{t-stamp}(N_i)$ is not the largest time stamp in G_u^m, has a successor N_k such that $\textit{t-stamp}(N_k) \geq \textit{t-stamp}(N_i) + 1$.

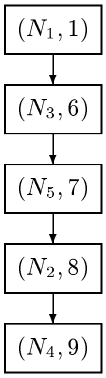

Fig. 5. A minimal unrolled graph

We define a function g such that $g(N_i)$ is the node of G with the smallest time-stamp larger than the time-stamp of N_i. It follows that for each edge $((N_i, t_j), (N_k, t_l)) \in E_u^m, N_k = g(N_i)$.

Observation 4.2. Let $N_k = g(N_i)$. If $\textit{t-stamp}(N_k) > \textit{t-stamp}(N_i) + 1$, then any node executed between the last execution of N_i and the last execution of N_k is either N_k or has been executed at least once after the last execution of N_k.

PROOF. Let N_m be a node executed between the last execution of N_i and the last execution of N_k. If $N_m \neq N_k$ and N_m has not been executed after the last execution of N_k then we would have $\textit{t-stamp}(N_i) < \textit{t-stamp}(N_m) < \textit{t-stamp}(N_k)$. Hence, $g(N_i) = N_m$ which is a contradiction. □

In the next section we prove an important property of G_u^m—that data flow analyses on G_u^m and G_u yield identical data flow information at a breakpoint for bit-vector data flow problems. This property enables us to perform dynamic currency determination by solving the data flow problem of *hoist-reach* (see Section 3) on G_u^m.

5. ADEQUACY OF THE MINIMAL UNROLLED GRAPH

The following data flow problem can be used over G_u^m to determine the dynamic currency of a variable V:

$$\textit{Hoist\_reach\_in}_i^V = \Sigma_p \; \textit{Hoist\_reach\_out}_p^V \tag{1}$$

$$\textit{Hoist\_reach\_out}_i^V = \textit{Hoist\_reach\_in}_i^V \cdot \neg \textit{Hoist\_kill}_i^V + \textit{Hoist\_gen}_i^V \tag{2}$$

where $\textit{Hoist\_gen}_i^V = \textit{true}$ if node N_i contains an occurrence of E_h^V not followed by an occurrence of E_r^V
$\textit{Hoist\_kill}_i^V = \textit{true}$ if node N_i contains an occurrence of E_r^V
$\cdot$, $+$ and $\neg$ are boolean operators and Σ_p is '+' over all predecessor nodes.

It is assumed that occurrences of E_h^V and E_r^V in the IR are identified from annotations introduced by the compiler. Variable V is dynamically current at a breakpoint

function	behavior	comment
START	$0 \mapsto 1$ $1 \mapsto 1$	constant at 1
STOP	$0 \mapsto 0$ $1 \mapsto 0$	constant at 0
PASS	$0 \mapsto 0$ $1 \mapsto 1$	passes argument along

Fig. 6. Functions in a bit-vector framework

situated at the beginning of node N_k if $Hoist\_reach\_in_k = false$ and dynamically noncurrent if $Hoist\_reach\_in_k = true$. Solving this data flow problem for the minimal unrolled graph of Figure 5, augmented with block $(Bkpt, \ldots)$ at the end yields $Hoist\_reach\_in^V_{Bkpt} = true$, which indicates that variable V is noncurrent at entry to the node containing the breakpoint.

Bit-vector data flow analysis

A monotone data flow analysis framework (or monotone framework) [Hecht 1977; Khedker and Dhamdhere 1994] is a triple $D = (L, \sqcap, F)$, where

(1) $(L, \sqcap)$ is a semilattice of finite height with a bottom element $\perp$.
(2) F is a monotone operation space associated with L.

A bit-vector framework is characterized by three properties—single bit representation of each data flow property, separability of solution and monotonic bit functions [Dhamdhere et al. 1992; Khedker and Dhamdhere 1994]. The second property implies that data flow properties can be evaluated independent of each other, i.e. they do not influence each other. Hence F consists of single-bit functions and the merge operation $\sqcap$ is a bit-wise AND or OR operation. The third property requires that a bit function cannot negate any bit. These properties together imply that $F = \{$START, STOP, PASS$\}$ as defined in Figure 6.

$I = (G, M)$ is an instance of a bit-vector framework [Hecht 1977; Khedker and Dhamdhere 1994], where

(1) $G = (N, E, s)$ is a flow graph.
(2) $M : N \to F$ is a function that maps each node in N to an operation in $F = \{$START, STOP, PASS$\}$.

It can be seen that the $Hoist\_reach\_in/Hoist\_reach\_out$ problem of Equations (1)-(2) is a bit-vector problem. The operation associated with a node N_i for a variable V is obtained from Equation (2) by substituting the values of $Hoist\_kill^V_i$ and $Hoist\_gen^V_i$. Figure 7 shows functions associated with the nodes of Figure 2 for determining the currency of variable V.

Note that the separability of solution required by bit-vector frameworks is violated by assignments through pointers. Hence the data flow analysis described here cannot be applied to assignments through pointers. For example, if E^V of Figure 2 is an assignment $*x := \ldots$ where x is a pointer, the functions associated with nodes $N4$ and $N5$ which contain E^V_h and E^V_r, respectively, would depend on assignments to x in the program. If node $N1$ contained as assignment $x := addr(a)$,

node	function for V
N_1	PASS
N_2	PASS
N_3	PASS
N_4	START
N_5	STOP
N_6	PASS

Fig. 7. Data flow functions for the program of Figure 2

node	function for a	function for b
N_1	PASS	PASS
N_2	PASS	PASS
N_3	PASS	PASS
N_4	START	PASS
N_5	STOP	PASS
N_6	PASS	PASS

(a)

node	function for a	function for b
N_1	PASS	PASS
N_2	PASS	PASS
N_3	PASS	PASS
N_4	PASS	START
N_5	PASS	STOP
N_6	PASS	PASS

(b)

Fig. 8. Data flow functions for Figure 2 for assignments through pointers

then $Hoist\_gen^a_4 = Hoist\_kill^a_5 = true$ and $Hoist\_gen^b_4 = Hoist\_kill^b_5 = false$ leading to the functions shown in Figure 8(a), whereas if node $N1$ contained as assignment $x := addr(b)$, then $Hoist\_gen^b_4 = Hoist\_kill^b_5 = true$ and $Hoist\_gen^a_4 = Hoist\_kill^a_5 = false$ leading to the functions shown in Figure 8(b).

Data flow analysis on G^m_u

Solving a data flow problem on G^m_u instead of G_u implies eliminating some data flow functions from the purview of data flow analysis. To prove adequacy of G^m_u for bit-vector data flow analysis, we show that the eliminated data flow functions are redundant for the purpose of bit-vector data flow analysis. To begin with, we define functions which help us to eliminate redundancies from a list of elements. We use these functions to reason about the elimination of data flow functions when we use G^m_u.

Function *last* takes a nonempty list as its argument and returns its last element. Function *lpp* takes a list as its argument and returns the largest proper prefix of the list (i.e. the list without its last element) if the list is nonempty and the empty list otherwise. Function *remove* takes a list and an item as its arguments and returns the list after removing all occurrences of the item from it. We use the operator $||$ to append an element to the end of a list. Function *reduce* on a list l returns a list that only contains the last occurrence of every element in l. The function is defined as follows:

$$reduce(l) = \begin{cases} \epsilon & \text{if } l = \epsilon \\ remove(reduce(lpp(l)),\ last(l)) \mathbin{||} last(l) & \text{otherwise} \end{cases}$$

We say that *reduce* returns a list which does not contain redundant elements. Function *compose* is defined as:

$$compose(f_1,\ \ldots,\ f_n) = f_n \circ\ \ldots\ \circ\ f_1$$

LEMMA 5.1. *If f and g are operations in a bit-vector framework, then:*

$$compose(f, g, f) = compose(g, f)$$

PROOF. The operations f, g can only be `START`, `STOP`, or `PASS`. The following equalities prove the lemma:

$$\forall\, x \in L :\ (\texttt{START}\ \circ\ g\ \circ\ \texttt{START})(x) = 1 = (\texttt{START}\ \circ\ g)(x)$$

$$\forall\, x \in L :\ (\texttt{STOP}\ \circ\ g\ \circ\ \texttt{STOP})(x) = 0 = (\texttt{STOP}\ \circ\ g)(x)$$

$$\forall\, x \in L :\ (\texttt{PASS}\ \circ\ g\ \circ\ \texttt{PASS})(x) = (\texttt{PASS}\ \circ\ g)(x)$$

□

LEMMA 5.2. *If l is a list of operations in a bit-vector framework,*

$$compose(l) = compose(reduce(l))$$

PROOF. We prove the lemma by induction on length of l. The basis is a list of a single element f:

$$compose(f) = f = compose(reduce(f))$$

Consider an arbitrary operation a. The induction step has two cases:
(1) $a \notin l$:

$$\begin{aligned}
compose(l \mid\mid a) &= a \circ\ compose(l) \\
&= a \circ\ compose(reduce(l)) \quad \text{by inductive hypothesis} \\
&= compose(reduce(l) \mid\mid a) \\
&= compose(reduce(l \mid\mid a)) \quad \text{because } a \notin l
\end{aligned}$$

(2) $a \in l$:

$$\begin{aligned}
compose(l \mid\mid a) &= a\ \circ\ compose(l) \\
&= a\ \circ\ compose(reduce(l)) \quad \text{by inductive hypothesis} \\
&\equiv a\ \circ\ f_1\ \circ\ \ldots\ \circ\ f_{i-1}\ \circ\ f_i\ \circ\ f_{i+1}\ \circ\ \ldots\ \circ\ f_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } a = f_i \\
&= a\ \circ\ f_1\ \circ\ \ldots\ \circ\ f_{i-1}\ \circ\ f_{i+1}\ \circ\ \ldots\ \circ\ f_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{by Lemma 5.1} \\
&= a\ \circ\ compose(remove(reduce(l),\ a)) \\
&= compose(remove(reduce(l),\ a) \mid\mid a) \\
&= compose(reduce(l \mid\mid a))
\end{aligned}$$

□

Consider a path of nodes $p = (i_1, \ldots, i_m)$. Let $M(i_j) = f_j \; \forall j$. Let x be the value of a data flow attribute at the beginning of i_1. For a forward data flow problem, the value of the data flow attribute at the end of p would be:

$$\begin{aligned}(f_m \circ \ldots \circ f_1)(x) &= \mathit{compose}(f_1, \ldots, f_m)(x) \\ &= \mathit{compose}(\mathit{reduce}(f_1, \ldots, f_m))(x)\end{aligned}$$

Let *e-path* be a projection function on an unrolled graph or minimal unrolled graph, such that $\textit{e-path}(G_u)$ is a list of nodes in which an occurrence of N_i precedes an occurrence of N_k (represented as $N_i \lessdot N_k$) iff $(N_i, t_j) \in G_u$, $(N_k, t_l) \in G_u$ and $t_j < t_l$.

LEMMA 5.3. Given a list of nodes p:

$$\mathit{reduce}(\textit{e-path}(G_u)) = \textit{e-path}(G_u^m)$$

PROOF. From part 1 of Definition 4.2, it is clear that $\mathit{reduce}(\textit{e-path}(G_u))$ and $\textit{e-path}(G_u^m)$ contain the same set of nodes. Consider two elements N_i and N_j of $\textit{e-path}(G_u)$. If $N_i \lessdot N_j$ in $\mathit{reduce}(\textit{e-path}(G_u))$, then the last occurrence of N_i in G_u precedes last occurrence of N_j. This implies that $\textit{t-stamp}(N_i) < \textit{t-stamp}(N_j)$. Hence $N_i \lessdot N_j$ in $\textit{e-path}(G_u^m)$ also. □

THEOREM 5.1. *Data flow analysis on an unrolled graph and its corresponding minimal unrolled graph yield the same results.*

PROOF. Let $\textit{e-path}(G_u) = (i_1, \ldots, i_m)$ and let $l = (M(i_1), \ldots, M(i_m))$ be the list of corresponding operations. We define a function M' to take a list of nodes and return the corresponding list of operations. Hence $l = M'(\textit{e-path}(G_u))$. Now,

$$\mathit{compose}(l) \quad = \quad \mathit{compose}(\mathit{reduce}(l)) \quad \text{by Lemma 5.2}$$

Substituting $l = M'(\textit{e-path}(G_u))$, we get

$$\begin{aligned}&\mathit{compose}(M'(\textit{e-path}(G_u))) \\ &= \mathit{compose}(\mathit{reduce}(M'(\textit{e-path}(G_u)))) \\ &= \mathit{compose}(M'(\mathit{reduce}(\textit{e-path}(G_u)))) \\ &= \mathit{compose}(M'(\textit{e-path}(G_u^m))) \qquad \text{by Lemma 5.3}\end{aligned}$$

which proves the theorem. □

6. USING THE MINIMAL UNROLLED GRAPH

The minimal unrolled graph G_u^m for an execution of program P can be constructed as follows: Code instrumentation (e.g., [Copperman and Thomas 1995]) can be used by the debugger to insert patch code at the start of each node of P. The patch code would save the time-stamp of that node[2]. At a breakpoint, the minimal unrolled graph is constructed by forming blocks of the form (N_i, t_j), where $t_j > 0$ is the time-stamp of N_i. The blocks are listed in ascending order by time-stamps and

[2]A goal of research in debugging of optimized code is to avoid code instrumentation as it de-optimizes a program by adding to its execution time. Hence a program may be run without instrumentation normally; it may be instrumented and re-run following an exception. However, such a debugger cannot deal with 'core' files.

adjacent blocks are joined with edges. Alternatively, MEPR can be constructed by listing the pairs (*nodeid*, *time-stamp*) in ascending order by time-stamps. At the end of the minimal unrolled graph we add a block (*Bkpt*, $lt+1$), where lt is the largest time-stamp associated with a node of G, to represent the node containing the breakpoint (note that the breakpoint is assumed to precede the first instruction in the node). Data flow analysis to determine the dynamic currency of a variable is now performed on the minimal unrolled graph. The time complexity of this analysis is $O(n)$, where n is the number of nodes in the flow graph, since the minimal unrolled graph consists of a single path in which each node occurs at most once.

As mentioned in Section 5, assignments through pointers cannot be handled by bit-vector data flow analyses. However, this does not prohibit dynamic currency determination in optimized programs which use pointers. The *Hoist_reach_in/out* data flow problem used for dynamic currency determination has a bit-vector data flow analysis framework. It determines the dynamic currency of a variable V based on annotations marking the presence of E_h^V and E_r^V in the IR. Hence ability to perform dynamic currency determination using the minimal unrolled graph merely depends on the ability of the compiler to insert annotations concerning E_h^V and E_r^V. The only limitations on dynamic currency determination would arise in situations involving the movement of pointer-based assignments—for example, the assignment $*x := \ldots$ where x is a pointer in Figure 8 of Section 5. However, such situations are very rare in practice.

The debugger needs to perform more work to determine the currency of a specific array element, e.g. $a[5]$, at a breakpoint. Apart from solving the *Hoist_reach_in/out* data flow problem using the annotations $E_h^{a[5]}$ and $E_r^{a[5]}$ in the IR, the debugger would have to determine the currency of all data entities of the form $a[k]$. $a[5]$ is noncurrent if some data entity $a[k]$ is noncurrent and $k=5$. If k itself is noncurrent at the breakpoint, the debugger would have to recover the current value of k using the techniques described in Section 6.1.

Run-time overheads

Code instrumentation necessary to collect time-stamps of nodes leads to moderate overheads. These were found to be in the range of 10–25 percent on a Sun Microsparc workstation with 32MB memory for routines from the Linpack library when each node of the program flow graph represented one source statement. The overheads can be reduced by using basic blocks as nodes of the program flow graph. Now, a breakpoint can occur inside a node rather than at its start, hence additional data flow analysis would be required *within* a node to determine whether occurrences of inserted or redundant expressions precede the breakpoint. Details of this analysis are obvious.

Run-time overheads can be further reduced by using superblocks (also called extended basic blocks [Muchnick 1997]) as nodes of the program flow graph. A superblock is a sequence of instructions which has a single entry point but may have multiple exit points. Thus a superblock may contain many basic blocks. Since the data flow function to be associated with a superblock depends on which exit is taken out of it, functions associated with different executions of a superblock are likely to be different. To overcome this difficulty, we define a function for each path in a superblock. (For simplicity, we associate these functions with different exits of a superblock.) Thus, a set of functions is associated with each superblock; the function to be used to represent a specific execution of a superblock would depend on the exit taken during that execution. The theory of Section 5 now applies to this function space. To facilitate building of the minimal unrolled graph, we now associate a time-stamp with each exit of a superblock. Use of superblocks reduces run-time overheads because time-stamping is performed when execution leaves a superblock rather than when execution enters or leaves each basic block in a superblock. Storage requirements are now $O(n_{exit})$ where n_{exit} is the total number of superblock exits in a program.

To identify larger superblocks, code duplication is used to eliminate entries into a sequence of instructions. Figure 9 shows superblock formation using code duplication. Basic block N_6 has been duplicated to form two superblocks SB_1 and SB_2 consisting of basic blocks N_1, N_2, N_3, N_6 and N_4, N_5, N_6'; respectively. A one-to-many function is now needed to map a source breakpoint into object breakpoints located in N_6 and N_6'. However, code duplication does not pose any problems in the use of minimal unrolled graphs; we simply associate identical functions with code copies. (For the same reason, loop unrolling [Muchnick 1997] does not pose any problems in dynamic currency determination of optimized programs.) Thus, if f_i is the function associated with basic block N_i, the functions associated with the two exits of SB_1 are f_1 and $f_6 \circ f_3 \circ f_2 \circ f_1$, respectively, while the function associated with the exit of SB_2 is $f_6 \circ f_5 \circ f_4$.

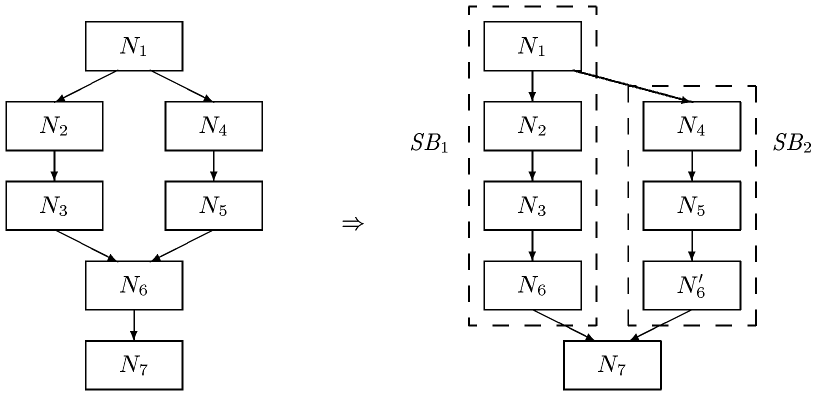

Fig. 9. Superblock formation by code duplication

Apart from the fact that run-time overheads can be reduced using basic blocks or superblocks, there is another reason why the overheads of code instrumentation need not be a deterrent to the practical use of minimal unrolled graphs: Code instrumentation needs to be performed by the debugger only when a program is to be debugged; normal executions of a program do not incur these overheads.

6.1 Other Applications of the Minimal Unrolled Graph

The equivalence of bit-vector data flow analyses on G_u^m and G_u is a general result which can be used for other purposes as well. Consider two data flow properties F_1 and F_2 which can be determined using forward bit-vector problems involving the merge operators Σ (as in Equation (1)), and Π, respectively. Let $F_1^{'}$ and $F_2^{'}$, respectively, be the values of these properties at a breakpoint as determined by performing data flow analyses on the minimal unrolled graph. Then $F_1^{'} = \mathit{true}$ implies $F_1 = \mathit{true}$, and $F_2^{'} = \mathit{false}$ implies $F_2 = \mathit{false}$. This property leads to obvious applications in determining whether a specific assignment of a variable V can influence its value at a breakpoint, and in detecting the use of uninitialized variables. We describe a not-so-obvious application of G_u^m in the following.

Minimal unrolled graph and dynamic slicing

The minimal unrolled graph can be used to aid in the use of a dynamic slice of a program P while debugging P. It can also be used instead of a dynamic slice in certain situations. We discuss these aspects in the following.

Definition 6.1. A *slice* (also called a *static slice*) of a program P with respect to a criterion $< p, V >$, where p is a program point and V is a variable, is an executable program P$'$ which contains a subset of statements in program P and which, on execution with any set of input values, yields the same value of V at p as program P.

Use of a slice reduces debugging effort; a bug in P which leads to an incorrect value of V at p must be present in the slice of P with respect to the criterion $< p, V >$. Note that a static slice may contain some redundancies from the viewpoint of a specific execution of P, since some statements in the static slice may not be visited during execution.

Definition 6.2. A *dynamic slice* of a program P with respect to a criterion $< inp, p, V >$, where inp is the set of values input to P, p is a program point and V is a variable, is an executable program P$''$ which contains a subset of statements in program P and which, on execution with input values inp, yields the same value of V at p as program P.

Note that a dynamic slice may contain fewer redundancies with respect to the execution of P for the set of input values *inp*. While debugging program P, the dynamic slice P$''$ is used to obtain the value of V at program point p. This is achieved by executing P$''$ until execution reaches point p. It can also be used to obtain a previous value of p by re-executing it till some point q which dynamically precedes point p. The dynamic slice is obtained by maintaining run-time information concerning the sequence of *all* program nodes visited during its execution before reaching point p [Korel and Laski 1988; Agrawal et al. 1993]. As such, G_u and EPR contain sufficient information for constructing a dynamic slice.

Use of a minimal unrolled graph can be combined with that of a dynamic slice as follows: A program P can be instrumented to obtain information concerning the program nodes visited until execution reaches point p. The minimal unrolled graph can now be used to determine the dynamic currency of variable V. If V is dynamically current, the minimal unrolled graph can yield the value of V without having

procedure *Recover_value*$(V, r, Bkpt)$
  Let $q := Previous\_defn(V, r)$;
  Let q contain an assignment $V := f_V(\ldots)$ such that RHS_V is
  the set of variables occurring in the RHS of the assignment.
  $\forall x \in RHS_V$
    if $Same\_value(x, Bkpt, q) = false$ **then**
      $Recover\_value(x, q, Bkpt)$;
  Execute the assignment $V := f_V(\ldots)$;
end

Fig. 10. Procedure *Recover_value*

to incur the overhead of building the dynamic slice. If variable V is noncurrent, a dynamic slice can be constructed using run-time information. (Note that this will require more run-time information than contained in G_u^m and MEPR.) This slice can be executed up to (but excluding) the node containing the occurrence of E_h^V which made V noncurrent at point p. The value of V at the end of this execution is the expected value of V at point p. Thus, we can recover the expected value of a dynamically noncurrent variable. In the following, we describe a more efficient approach to recovery using G_u^m.

Recovery of dynamically noncurrent scalar variables

In certain situations, a minimal unrolled graph can be used to recover the value of a dynamically noncurrent scalar variable in an optimized program. This is far more efficient than the use of a dynamic slice. We assume all assignments to scalar variables to be of the form $x := f_x(\ldots)$, where f_x is a function of scalar variables. We define the following for the purpose of recovery: Function $Previous\_defn(V, p)$ returns the identity of the program point at which the last assignment to V preceding point p in G_u^m occurs. Predicate $Same\_value(V, p, p')$, where point p' precedes p in G_u^m, indicates whether the actual value of variable V at p is the same as its value at p'. Note that $Same\_value(V, p, p')$ would be *false* if an assignment to V was performed between p' and p.

At a breakpoint *Bkpt*, procedure $Recover\_value(V, r, Bkpt)$ recovers the value of V according to the last assignment to V preceding point r in G_u^m. If E_h^V occurring at point p_h makes V dynamically noncurrent at *Bkpt*, we recover the expected value of V by making the procedure call $Recover\_value(V, p_h, Bkpt)$. Note that recovering the value of V according to an assignment may require us to recover the values of variables occurring on the right hand side of the assignment. Figure 10 shows the recovery algorithm used by procedure *Recover_value*. We use the notion of shadow variables to enable resumption of a program's execution following recovery. Thus, to recover the value of V at *Bkpt*, a shadow variable x' is created for every variable x in the program. All recovery actions are performed on shadow variables. After executing $Recover\_value(V, p_h, Bkpt)$, the value of V' is reported to the user.

Restrictions on the correctness of the recovery procedure arise from two sources—loop structure and the nature of assignments in a program. Consider two statements S_1 and S_2 located in nodes N_1 and N_2, respectively, of P. Let blocks (N_1, t_j) and (N_2, t_l) be consecutive blocks in G_u^m and let $t_l > t_j + 1$. From Observation 2 of

Section 4, it is clear that any nodes executed between these instances of execution of N_1 and N_2 must have been executed again. This implies existence of loop(s) in the program. In the absence of information concerning loop structure of the program (that is, identities of nodes in each loop) and the order in which program nodes were visited during execution, we have to assume that any node with a time stamp $> t_j$ in G_u^m could have been executed between the last executions of nodes N_1 and N_2. This fact leads to some restrictions as described in the following.

Consider nodes N_i and N_j containing points q_i and q_j, respectively, such that blocks (N_i, t_i) and (N_j, t_j) occur in G_u^m. Let procedure call *Recover_value*$(V, p_h,$ *Bkpt*$)$ make a recursive call *Recover_value*$(x, q_j,$ *Bkpt*$)$ during the recovery of V, and let $q_i =$ *Previous_defn*(x, q_j). Let an assignment $x := \ldots z \ldots$ occur at q_i and *Same_value*$(z,$ *Bkpt*$, q_i) =$ *false*. If there exist consecutive blocks (N_l, t_l) and (N_m, t_m) along the path segment $(N_k, t_k) \ldots (N_i, t_i)$ in G_u^m such that $t_m > t_l + 1$ and node N_k contains the last assignment of z in G_u^m prior to point q_i then procedure *Recover_value* recovers the correct value of V only if no assignments to z occur in any block (N_g, t_g) such that $t_g > t_k$. It is therefore necessary for procedure *Recover_value* to abandon recovery if it finds that an assignment violating this condition exists in the program. As an example, consider the program of Figure 2 and the minimal unrolled graph of Figure 5. If assignment $x := \ldots z \ldots$ exists at point q_i in block $(N_3, 6)$, *Same_value*$(z,$ *Bkpt*$, q_i) =$ *false* and block $(N_1, 1)$ contains an assignment to z, then recovery must be abandoned if any of the nodes N_2, N_3, N_4 or N_5 contain an assignment to z.

Another restriction on the functioning and correctness of procedure *Recover_value* arises from the presence of pointers. While defining predicate *Same_value*(V, p, p'), we have assumed that the debugger will be able to recognize *all* definitions of V located between p' and p. This will require the debugger to possess information about aliases (possibly induced by assignments through pointers) [Aho et al. 1986]. In the absence of such information, it is conservative to assume *Same_value* to be *false* if the debugger encounters *any* assignment through a pointer while looking for an assignment to V. However, such conservative behavior is not possible in function *Previous_defn*(V, r). In the absence of aliasing information, this function will have to abort recovery if it encounters any assignment through a pointer before finding an assignment of the form $V := \ldots$ in a backward scan of G_u^m starting at point r.

Programs containing function and procedure calls can be handled as follows: If superblocks are being used, the debugger ensures that time-stamping is performed before invoking a function or procedure. A program containing parameterless functions and procedures is handled analogous to a program without function or procedure calls. The restrictions described earlier apply directly to such programs. For functions and procedures with parameters, extra annotations would have to be inserted by a compiler. For a call $P(x)$ where P has a formal parameter g, an annotation g '*assign*' x may be inserted preceding the code for the call $P(x)$. If this annotation is encountered while recovering the value of g, the debugger would treat it as an assignment $g := x$. If g is a value parameter, this reflects the exact behavior of the program at the call $P(x)$. For a reference parameter, this has the effect of a 'name translation'—instead of locating a previous definition of g, the debugger would now locate a previous definition of x. If a function or procedure is invoked more than once during the execution of a program, it leads to a situation analogous to the presence of loops. Hence recovery in programs containing multiple invocations of a procedure attracts the restriction arising from Observation 2 mentioned earlier.

Recovery of dynamically noncurrent subscripted variables

Consider an assignment $a[i] := \ldots x \ldots b[j] \ldots$ occurring at point q in procedure $Recover\_value(a[i], p_h, Bkpt)$. To recover the value of $a[i]$ according to this assignment, in procedure *Recover_value* we proceed as if the assignment is of the form $a := f_{a[i]}(i, \ldots x, \ldots j, b[j])$. This approach will ensure that i, j and $b[j]$ will have appropriate values when we compute the previous value of $a[i]$. Compared to the recovery of scalar variables, two new issues arise in such recovery. First, predicate $Same\_value(b[j], Bkpt, p')$ will need to decide whether the value of $b[j]$ has changed between points p' and $Bkpt$. This will require examination of all assignments of the form $b[k] := \ldots$ for some k to determine whether $b[k]$ and $b[j]$ refer to the same element of b. As in the recovery of scalar variables, *Same_value* should be *false* if the debugger is unable to determine whether the value of $b[j]$ has changed between points p' and *Bkpt*. Second, while recovering the value of $b[j]$ preparatory to execution of the assignment $a[i] := \ldots x \ldots b[j] \ldots$, procedure *Recover_value* should abandon the recovery of $a[i]$ if it encounters an assignment to $b[k]$ for some k and cannot determine whether $b[k]$ and $b[j]$ refer to the same element of b.

7. CONCLUSIONS

The minimal unrolled graph is an efficient representation of an execution of a program. Its space requirements are $O(n)$ where n is the number of nodes in a program flow graph. Code instrumentation necessary to collect time-stamps of nodes leads to moderate overheads. These overheads can be reduced using basic blocks or superblocks as nodes of the program flow graph.

The overheads of code instrumentation need not be a deterrent to the practical use of minimal unrolled graphs, since code instrumentation needs to be performed by the debugger only when a program is to be debugged; normal executions of a program do not incur these overheads. Thus, the minimal unrolled graph is a practical basis for providing a new set of features in the debugging of optimized programs, viz. dynamic currency determination and recovery of noncurrent variables subject to the limitations mentioned in Section 6.

ACKNOWLEDGMENTS

Authors wish to thank the referees for many useful suggestions, and A. V. V. S. Kiran for discussions concerning dynamic slices.

REFERENCES

Adl-Tabatabai, A. 1996. Source-Level Debugging of Globally Optimized Code. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Adl-Tabatabai, A. and Gross, T. 1993a. Detection and Recovery of Endangered Variables Caused by Instruction Scheduling. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, *28*, 6, 13–25.

Adl-Tabatabai, A. and Gross, T. 1993b. Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging. In *Proceedings of the 20th Annual ACM SIGACT-*

SIGPLAN Symposium on Principles of Programming Languages. ACM, New York, Charleston, South-Carolina, 371–383.

Adl-Tabatabai, A. and Gross, T. 1996. Source-Level Debugging of Globally Optimized Code. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, *31*, 5, 13–25.

Agrawal, H., DeMillo, R. A., and Spafford, E. H. 1993. Debugging with dynamic slicing and backtracking. *Software—Practice & Experience 23*, 6, 589–616.

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers—Principles, Techniques and Tools*. Addison Wesley.

Copperman, M. 1993. Debugging Optimized Code Without Being Misled. Ph.D. thesis, Computer and Inforamation Sciences, University of California at Santa Cruz, Santa Cruz, California.

Copperman, M. 1994. Debugging Optimized Code Without Being Misled. *ACM Transactions on Programming Languages and Systems*, *16*, 3, 387–427.

Copperman, M. and Thomas, J. 1995. Poor Man's Watchpoints. *ACM SIGPLAN Notices*, *30*, 1, 37–44.

Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. 1992. How to Analyze Large Programs Efficiently and Informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, *27*, 7, 212–223.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc.

Hennessy, J. 1982. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, *4*, 3, 323–344.

Khedker, U. P. and Dhamdhere, D. M. 1994. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Transactions of Programming Languages and Systems*, *16*, 5, 1472–1511.

Korel, B. and Laski, J. 1988. Dynamic program slicing. *Information Processing Letters*, 29, 155–163.

Muchnick, S. 1997. *Advanced Compiler Design & Implementation*. Morgan Kaufmann.

Wismüller, R. 1994. Debugging of Globally Optimized Programs Using Data Flow Analysis. *ACM SIGPLAN Notices*, *29*, 6, 278–289.

Zellweger, P. T. 1983. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*. ACM SIGPLAN Notices, *18*, 8, 159–172.

Received February 1997; revised March 1998; accepted July 1998