Debugging Optimized Code Without Being Misled

MAX COPPERMAN
University of California at Santa Cruz

Correct optimization can change the behavior of an incorrect program; therefore at times it is necessary to debug optimized code. However, optimizing compilers produce code that impedes source-level debugging.

Optimization can cause an inconsistency between where the user expects a breakpoint to be located and the breakpoint's actual location. This article describes a mapping between statements and breakpoint locations that ameliorates this problem. The mapping enables debugger behavior on optimized code that approximates debugger behavior on unoptimized code sufficiently closely for the user to use traditional debugging strategies.

Optimization can also cause the value of a variable to be *noncurrent*—to differ from the value that would be predicted by a close reading of the source code. This article presents a method of determining when this has occurred, and shows how a debugger can describe the relevant effects of optimization. The determination method is more general than previously published methods; it handles global optimization and many flow graph transformations, and it is not tightly coupled to optimizations performed by a particular compiler. Necessary compiler support is also described.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*

General Terms: Performance, Verification

1. INTRODUCTION

1.1 The Problem

A source-level debugger has the capability of setting a breakpoint in a program at the executable code location corresponding to a source statement. When a breakpoint at some point P is reached, a debugger user can query the value of a variable V, and the debugger will display the value in V's storage location. Optimization may have reordered, rearranged, or even suppressed assignments to V, making the value in V's storage location misleading. The debugger user may expend time and effort attempting to

This work was supported in part by MetaWare Incorporated and in part by IBM grant SL88096.
Author's address: Computer and Information Science, University of California at Santa Cruz, Santa Cruz, CA 95064.

determine why V contains the value that has been displayed when the source code suggests that V should contain some other value.

Figure 1 is an example in which constant propagation followed by dead-store elimination creates the potential for the user to be misled. Assume that the only use of `x` following the assignment of `constant` to `x` is the assignment of `x` to `y`. Constant propagation removes that use of `x` as shown in the second column of the figure. With that use eliminated, the assignment of `constant` to `x` may be eliminated, as shown in the third column. If a breakpoint is reached anywhere following the eliminated assignment to `x`, and the debugger is asked to display the value of `x`, then typical debuggers will display `expression`. The user, looking at the original source code, may be confused by the fact that the displayed value is not `constant`, or may believe wrongly that the value being assigned to `y` is `expression`. At such a breakpoint, `x` is called *noncurrent* [Hennessy 1982], and determining whether optimization has caused a variable's value to be misleading is called *currency determination*.

Optimization may also introduce confusion over where execution is suspended in the program being debugged. The straightforward mapping of statement boundaries onto machine code locations in unoptimized code is insufficient for optimized code.

Zellweger [1984] introduced terms for two methods of removing or ameliorating the confusion introduced into the debugging process by optimization. One method, known as providing *expected* behavior, is to have the debugger responses to queries and commands on an optimized version of a program be identical to its responses to the same set of queries and commands on an unoptimized version of the program. It may not always be possible to provide expected behavior, so the next best thing is to provide *truthful* behavior, in which the debugger avoids misleading the user, either by describing in some fashion the optimizations that have occurred or by warning the user that it cannot give a correct answer to the command or query.

The misleading effects of optimization may sometimes be avoided by disabling optimization when debugging. This may be impractical, and at best it is inconvenient, because it requires extra compilation steps. At worst, however, it may be impossible. A program compiled with optimization enabled may behave differently from the same program compiled with optimization disabled—that is, when optimization is turned off, the bug may go away.

1.1.1 *Optimization May Change the Behavior of a Program.* There are two circumstances in which correct optimization may change the behavior of a program.[1]

[1]If program behavior changes because optimization is incorrect, there are three options. get a different compiler, get the broken compiler fixed, or work around the bug. In practice the first two options may not be viable. The third option requires the programmer to find the code that causes the compiler bug to show up and replace it with semantically equivalent code on which the compiler functions correctly. The programmer still has to debug the (incorrectly) optimized code! Even if the choice is made to get the compiler fixed, the programmer typically has to debug the optimized code enough to convice the compiler vendor that it is a compiler bug.

Original Source Code	After Constant Propagation	After Dead-Store Elimination
`x = expression;`	`x = expression;`	`x = expression;`
`...`	`...`	`...`
`x = constant;`	`x = constant;`	
`...`	`...`	`...`
`y = x;`	`y = constant;`	`y = constant;`
`...`	`...`	`...`

Fig. 1. Potentially confusing optimizations.

```
void uses_uninitialized_variable() {

    int x,y;

    return y;

    }
```

Fig. 2. Optimization can change program behavior: The stack location for y changes depending on whether storage is allocated for x, which in turn depends on whether the program is optimized. Because y is used without being initialized, that can change the return value of the function.

Loose Semantics. A language may contain constructs whose semantics allow multiple correct translations with distinct behaviors. Most common general-purpose programming languages do contain such constructs. The most commonly known area of "loose semantics" is evaluation order.

Buggy Programs. Optimizations are correctness-preserving transformations, but correctness-preserving transformations are not guaranteed to preserve the behavior of an incorrect program. This is commonly overlooked but important because a program that is being debugged is unlikely to be correct.

There is a widespread misconception that if optimization changes the behavior of a program, the compiler must be incorrect. Figure 2 presents an incorrect program whose behavior can be changed by correct optimization. It is a simplified example of a very common bug: using a variable that is not properly initialized. When `uses_uninitialized_variable` is called, a stack frame is allocated for it, within which **x** and **y** are located. **0** may be in the stack location allocated to **x**, and **1** may in the stack location allocated to **y**—**1** is returned. Because **x** is not used, its storage may be optimized away. In the optimized version, **y** is located where **x** falls in the unoptimized version —and now **0** is returned.

Because optimization can change the behavior of a program, it is necessary, upon occasion, either to debug optimized code or never optimize the code. This paper presents a mapping between statements and breakpoint locations that enables debugger behavior on optimized code to approximate debugger behavior on unoptimized code sufficiently closely for the user to use traditional debugging strategies.

Using this mapping, this work presents a currency determination technique that determines whether a variable has the value the user would expect when execution is suspended at one of these breakpoints. In response to a query about a variable *V*, this technique enables a debugger to

—display *V*'s value without comment if optimization has not given it a misleading value,

—display *V*'s value with a warning if optimization may have given it a misleading value. The warning can describe how the variable may have gotten the misleading value. The debugger can distinguish the cases in which *V* is known to have an unexpected value from the cases in which (because of unknown control flow) it may or may not, and adjust the warning accordingly.

1.2 Related Work

General approaches to debugging optimized code have been:

—To restrict the optimizations performed by the compiler to those that do not provoke the problem [Warren and Schlaeppi 1978; Zurawski and Johnson 1990]. This has the drawback that it degrades the efficiency of software compiled with such a compiler.

—To restrict the capabilities of the debugger to those that do not exhibit the problem [Warren and Schlaeppi 1978; Gupta 1990; Zurawski and Johnson 1990]. This is clearly undesirable, though possibly preferable to being misled by the debugger.

—To recompile, without optimization, during an interactive debugging session, the region of code that is to be debugged [Feiler and Medina-Mora 1980; Zurawski and Johnson 1990]. This requires a software engineering environment that provides incremental compilation. Such environments are not in general use, and even should they become commonplace, the approach is problematic because optimization may change the behavior of the program.

—To have the compiler provide information about the optimizations that it has performed and to have the debugger use that information to provide appropriate behavior [Adl-Tabatabai and Gross 1993b; Berger and R. Wismüller 1993; Brooks et al. 1992; Cohn 1991; Cool 1992; Copperman 1992; Copperman and McDowell 1991; Coutant et al. 1988; Gupta 1990; Hennessy 1982; Pineo and Soffa 1991; Pollock and Soffa 1992; Srivastava 1986; Warren and Schlaeppi 1978; Zellweger 1984; Zurawski and Johnson 1990]. A drawback of this approach is that it increases compilation time and symbol table size. Experience in debugging unoptimized code has shown that the benefits of source-level debugging are worth the time and space cost.

My work follows the fourth approach. Some previous work that has taken this approach has resulted in compiler/debugger pairs that are able to provide acceptable behavior when debugging optimized code because the debuggers have been specialized to handle the particular optimizations performed by the compiler. Because much of the industry allows compilers and debuggers to be mixed and matched, solutions that do not require the compilers and debuggers to be tightly coupled are preferable. This article describes a compiler/debugger interface for currency determination that does not require that the debugger be specialized to a particular set of optimizations.

Hennessy [1982] introduced the problem of currency determination. He presents a solution for local optimization, assuming either that no block in the unoptimized code contains more than one assignment to a given variable or that if it does, only the last such assignment in a block assigns to the location that the debugger associates with the variable [Copperman and McDowell 1993]. In 1982 this was not a restrictive assumption, but today many debugging-information formats allow the location associated with a variable to be a register. Hennessy also introduced the concept of *recovery*, in which the debugger reconstructs the value a variable should have at some point. Coutant et al. [1988] describes a debugger that solves several problems related to debugging optimized code for the particular set of optimizations performed by their compiler. Much of their work applies beyond that set of optimizations, but their solution to currency determination applies only to local optimization. Streepy [1991] and Brooks et al. [1992] present a breakpoint model designed for optimized code with the intent of helping the user understand the effects of optimization as opposed to hiding those effects. Their model is largely orthogonal to the one presented herein, and a combination would work well. Cool's [1992] work also aims at helping the user understand the effects of optimization by using different levels of highlighting for expressions that have not been executed, that have begun execution, and that have completed execution, for VLIW code after instruction scheduling. Berger and Wismüller [1993] also use dataflow techniques to attack currency determination. Their breakpoint model is more general, and they incorporate some aspects of recovery into their currency determination algorithm. As of this writing, their work is promising but incomplete. No previous work explicitly addresses optimizations that change the shape of a program's flow graph.

1.3 Overview of the Solution

The fundamental idea behind the solution to the currency determination problem is the following: if the definitions of a variable V that "actually" reach a point P are not the ones that "ought" to reach P, V is not current at P. The definitions of V that actually reach P are those that reach P in the version of the program executing under debugger control. The definitions of V

that ought to reach P are those that reach P in a strictly unoptimized version of the program.

A mapping between source statements and (optimized) executable code grounds the concept of "a point in a program."

Data-flow analysis on a graph data structure that combines preoptimization and postoptimization information finds the definitions that actually reach a point and those that ought to reach that point. This data structure must be modified when optimization changes the shape of the program flow graph, in order to combine this information correctly. Pointers add considerable complexity to the data flow analysis, so a solution that works in the absence of pointers is presented first, then extended to handle pointers.

The work in this article is extracted from my dissertation. The breakpoint model is similar to that in my earlier work, but the currency determination technique differs. In particular, in earlier work [Copperman and McDowell 1991], two reaching-definitions computations were performed independently, one on a preoptimization flow graph, and one on a postoptimization flow graph, and the results were compared. Because the computations were independent, a further computation (a graph search) might be required. In this work, a single reaching-definitions computation is performed on a data structure that incorporates both preoptimization and postoptimization information, eliminating the need for further computation. Copperman [1992] is a workshop paper that has one foot in each camp.

This work is applicable in the presence of any sequential optimizations that either do not modify the flow graph of the program or modify the flow graph in a constrained manner. Blocks may be added, deleted, coalesced, or copied; edges may be deleted, but control flow may not be radically changed. Allowable flow graph transformations are summarized in Section 4.3; they are described in detail in Appendix B and in Copperman [1993]. The techniques presented in this article apply in the presence of local common-subexpression elimination, global common-subexpression elimination, constant and copy propagation, constant folding, dead-code elimination, dead-store elimination, cross-jumping, local instruction scheduling, global instruction scheduling, strength reductions, code hoisting, partial-redundancy elimination, other code motion, induction-variable elimination, loop unrolling, and inlining (procedure integration), as well as any other optimizations that observe the constraints. As an example of an optimization that modifies control flow in a manner that does not observe the constraints, the techniques do not apply to a portion of a program that has had loops interchanged.

2. BREAKPOINT MODEL

In an unoptimized translation of a program, code is generated for every source code statement in the order in which it appears in the source code, and the code generated from most statements is contiguous.[2] It is possible to halt unoptimized code at a point that corresponds exactly to a statement boundary in the source code by halting at (before execution of) the first instruction generated from the statement. When execution is suspended at statement S

in unoptimized code, all "previous" statements have completed, that is, all code that was generated from statements on the path to S has been executed. No "subsequent" statements have begun, that is, no code that was generated from any statement on the path from S (including code generated from S itself) has been executed. Because of the straightforward nature of the translation, the value in each variable's location matches the value of the variable that would be predicted by a close reading of the source code.

In general, there is no point in optimized code that corresponds exactly to a statement boundary in the source code, that is, there is no point at which optimized code can be halted such that the above characteristics hold. Some choice of breakpoint location must be made nonetheless. The choice of breakpoint location is not crucial to the correctness of the work presented in the remainder of the article. It *is* crucial that the relative ordering of variable definitions and breakpoints be available, and that breakpoints be at locations that exist in both the optimized and unoptimized versions of a program. We describe one satisfactory breakpoint model which we use throughout the article.

Zellweger [1984] introduced the terms *syntactic* and *semantic* breakpoints. The order in which syntactic breakpoints are reached reflects the syntactic order of source statements; the syntactic breakpoint for statement n is prior to or at the same location as the syntactic breakpoint for statement $n + 1$. If the code generated from statement n is moved out of a loop, a syntactic breakpoint for n remains inside the loop. The semantic breakpoint location for a statement is the point at which the action specified by the statement takes place. This does not preserve any particular order. If the code generated from statement S is not contiguous, the semantic breakpoint location depends on the definition of "the action specified by the statement." If no code motion or elimination has occurred, the syntactic and semantic breakpoint for a statement are one and the same.

Streepy [1991] and Brooks et al. [1992] describe a source code/breakpoint location mapping that allows breakpoints to be set at various levels of granularity, including expressions, basic blocks, and subroutines. In the debugger described by Streepy and by Brooks et al., a breakpoint is set at the beginning of each sequence of contiguous instructions generated from the specified source construct.

The mapping presented below differs in that when a breakpoint is set on a statement, it allows the debugger to break once each time a statement is executed. It is complementary to the mapping presented by Streepy and Brooks et al., and they could be combined to good effect.

The view taken in this work is that the best semantic breakpoint location for a statement is the address of the instruction that most closely reflects the

[2]Code generated from looping or branching statements is typically not contiguous. However, this lack of contiguity is present in the source code as well as in the generated code. It can cause debugging anomalies in unoptimized code. For example, placing a breakpoint at a C **for** loop can cause several commonly available debuggers to either break once before loop entry or break each time through the loop, depending on the presence or absence of initialization code.

effect of the statement on user-visible entities (program variables and control flow). For statements involving program–variable updates, it is the instruction that stores into the variable. (A "store" in this context needs not be a store into a memory location. It can be a computation into a register, or a register copy, if that is the instruction that implements the semantics of the assignment.) For control flow statements (branching or looping constructs), it is the instruction that accomplishes the control transfer (typically a conditional branch), because it provides a natural sequence point for program dependences. An instruction whose address is the breakpoint location for a statement S is the *representation instruction* for S. The C statement

```
if ((i=j++) ==k)
```

has three representative instructions (and therefore three breakpoint locations), one at the store into `j`, one at the store into `i`, and one at the branch to the **then** or **else** case.

The syntactic breakpoint location for a statement S whose representative instruction has been moved or eliminated is the (identically syntactic and semantic) breakpoint location for the next statement in the same basic block whose representative instruction has not been moved or eliminated. If there is none, the syntactic breakpoint location for S is the last representative instruction in the block. If the entire block has been eliminated, S has no syntactic breakpoint location. The results described in the remainder of this article hold at syntactic breakpoints, because they are locations that are guaranteed to exist in both the optimized and unoptimized versions of a program.

3. CURRENCY

When the user asks the debugger to display the value of a variable, the user is misled if optimization has caused the value displayed to be different from the value that would be predicted by examing the source code.

The *actual value* of a variable V when execution is suspended at a breakpoint is the value in V's storage location. A variable's *expected value* when execution is suspended at a breakpoint is the value that would be predicted by examining the source code and knowing the relevant context, such as within which iteration of a loop execution is suspended.

In unoptimized code, at each breakpoint the expected value of every variable is identical to its actual value. In optimized code, the actual value of a variable at some point may differ from its expected value at that point. Hennessy [1982] introduced the terms *current*, *noncurrent*, and *endangered* to describe the relationship between a variable's actual value and its expected value at a breakpoint. This relationship is described on the basis of a static analysis, one that has no information about how the breakpoint was reached.

Examples of current, noncurrent, and endangered variables are presented. All examples use program flow graphs, where nodes represent basic blocks, and edges represent block connectivity. For clarity of exposition, the examples are minimal. The language of the examples includes assignment (`a = x`

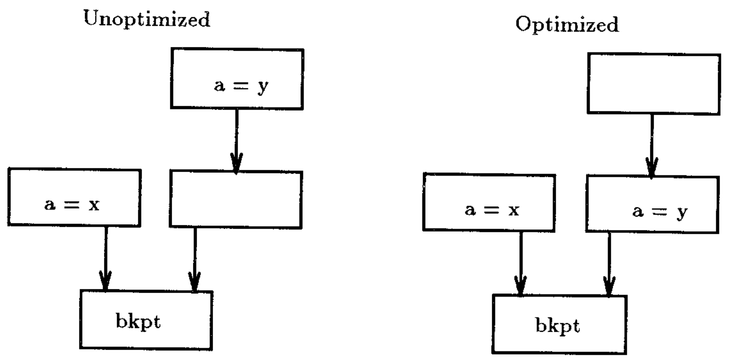


Fig. 3. Variable a is current at bkpt in the presence of relevant optimization.

denotes the assignment of x into a) and a distinguished symbol bkpt which represents the instruction at which the breakpoint has been reached. Assignment instructions with the same right-hand side assign the result of the equivalent computations into the left-hand side; this is how the relationship between assignments in the unoptimized code and assignments in the optimized code is shown. While a statement in a source language that corresponds to either an assignment or a breakpoint may compile to more than a single machine instruction, assignments and breakpoints appearing in flow graphs are referred to as instructions, because a single representative instruction is chosen for each statement.

Informally, a variable V is *current* at a breakpoint B if its actual value at B is guaranteed to be the same as its expected value at B no matter what path was taken to B.

A variable may be current at a breakpoint even if optimization has affected assignments into the variable. Figure 3 shows a case in which an assignment into a has been moved. Variable a is current at bkpt, because the code motion has not changed the fact that along each path a receives its value from the same assignment in the unoptimized and optimized versions of the program.

V is *noncurrent* at B if its actual value at B may differ from its expected value at B no matter what path is taken to B (though the two values may happen to be the same on some particular input). Figure 4 is a simple example of a noncurrent variable, and could be a result of dead-store elimination (or of code motion into a block not shown). There is only one way to reach bkpt in both versions of the program. There is a single assignment into a prior to the breakpoint in the unoptimized code, but in the optimized code there is no corresponding assignment into a along the only path to bkpt.

V is *endangered* at B if there is at least one path to B along which V's actual value at B may differ from its expected value at B. Endangered includes noncurrent as a special case.

In Figure 5, along the left-hand path the assignment into a that reaches bkpt in the unoptimized code corresponds to the assignment into a that

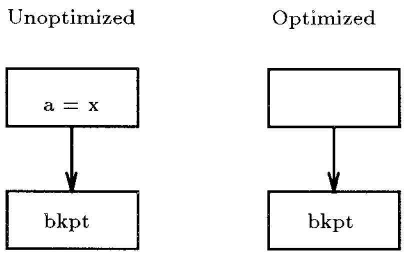


Fig. 4. Variable a is noncurrent at bkpt.

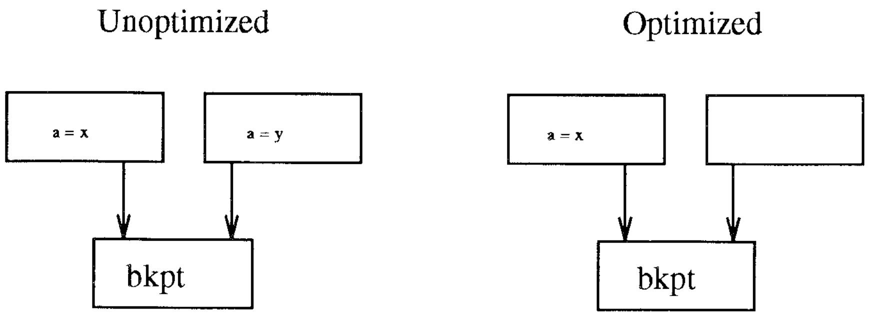


Fig. 5 Variable a is endangered at bkpt.

reaches bkpt in the optimized code, but along the right-hand path there is no such correspondence; a is endangered by virtue of the right-hand path, and is not noncurrent by virtue of the left-hand path.

The use of the terms current and noncurrent extends to particular paths: in Figure 5, a is current along the left-hand path and noncurrent along the right-hand path. When execution is suspended at bkpt during some particular run of the program, a is either current or noncurrent, depending on the path taken to bkpt. However, static analysis can determine only that a could be current or noncurrent, because knowledge of the path taken is absent. A debugger that has no access to execution history information can do no better than static analysis. Complete information about the execution path taken could be large, and collecting it could be invasive and time consuming. An open problem, termed *dynamic currency determination*, is how a debugger can collect the minimal information needed to determine whether an endangered variable is current or noncurrent when execution is suspended at a breakpoint. We assume such information is unavailable to the debugger.

In order to talk about V's currency along a particular path, a path must be defined in such a way that it makes sense in both the unoptimized and optimized versions of the program, as optimization may modify the program's flow graph.

Definition 1. A *path-pair* p is a pair $\langle p_u, p_o \rangle$ where p_u is a path through the flow graph of an unoptimized version of a program, and p_o is a path through the flow graph of an optimized version of the same program such that p_u and p_o are taken on the same set of inputs.

Because a path describes an entire execution, call blocks are expanded. A call block appears on a path, followed by the blocks visited in the called subroutine, followed by the call block's successor.

I have been using the term "unoptimized version" as if there were a canonical unoptimized translation of a program, and similarly I have used the term "optimized version" as if there were a canonical optimized translation. Of course, there are no such canonical translations. What is necessary is that there be a mapping between these "versions." The nature of the mapping will be discussed in Section 4.3. For the purposes of this section, simply assume all versions are produced by the same (correct) compiler, which has the same information available to it whether producing an unoptimized version or an optimized version. An implementation would most likely use a single compilation to produce unoptimized intermediate code, providing all necessary knowledge about what I refer to as the unoptimized version without actually generating machine code from it. It would then optimize that intermediate code to produce the optimized version.

Parts of a path-pair are of interest, i.e., a path-pair to a breakpoint or a path-pair from one point to another.

Definition 2. A *path-pair p to a block B*, where B is visited in both versions, is a sub-path-pair of a path-pair p' where p_u is a prefix of p'_u ending in an occurrence of B, and p_o is a prefix of p'_o ending in the same occurrence of B.

Definition 3. A *path-pair p from block A to block B*, where A and B are visited in both versions, is a sub-path-pair of a path-pair p' where p_u is a subsequence of p'_u starting at an occurrence of A and ending at an occurrence of B, and p_o is the subsequence of p'_o starting at the same occurrence of A and ending at the same occurrence of B.

I speak loosely of a path-pair to a breakpoint, or a path-pair from one representative instruction to another. In these cases, I mean a path-pair to the block containing the breakpoint, or from the block containing one representative instruction to the block containing the other.

I make a simplifying assumption that a variable resides in a single location throughout its lifetime. Relaxing this assumption is a topic for future research. Both assignments to a variable and side effects on that variable modify the value stored in that variable's location. These terms do not distinguish whether the source code or generated code is under discussion. Furthermore, they do not distinguish between unoptimized and optimized generated code. These distinctions are needed because we compare reaching definitions computed on unoptimized code with reaching definitions computed on optimized code. Henceforth the term *assignment* refers to assignments and side effects in the source code.

It is convenient to have a term *definition* that can denote either an assignment or its representative instruction in unoptimized code. This does not introduce ambiguity because either one identifies the other, and the order of occurrence is the same in the source code and unoptimized code generated from it. In contrast, the term *store* denotes a representative instruction for an assignment in optimized code. As with definitions, a store has a corresponding assignment, but unlike definitions, an assignment may have no corresponding store; and the order of occurrence of stores in the machine code may differ from the order of occurrence of assignments in the source code.

An optimizing compiler may be able to determine that two assignments to a variable are equivalent and produce a single instance of generated code for the two of them, or it may generate multiple instances of generated code from a single assignment. Such optimizations essentially make equivalent definitions (or stores) indistinguishable from one another. We will be concerned with determining whether a store that reaches a breakpoint was generated from a definition that reaches the breakpoint. If definitions d and d' are equivalent, and store s was generated from d while s' was generated from d', the compiler is free to eliminate s' so long as s reaches all uses of d'. To account for this, s needs to be treated as if it was generated from either d or d'.

Definition 4. A *definition of* V is an equivalence class of assignments to V occurring in the source code of a program that have been determined by a compiler to represent the same or equivalent computations, or the set of representative instructions generated from members of such an equivalence class in an unoptimized version of the program.

Definition 5. A *store into* V is the set of representative instructions occurring in an optimized version of a program that were generated from any member of the equivalence class denoted by a definition.[3]

We can now formally define some of the terms described previously. The following definitions assume that the breakpoint location is the same in the optimized and unoptimized version, that is, either that the representative instruction for the statement at which the breakpoint is set has not been moved by optimization or that a syntactic breakpoint has been specified.

The definition of "current" is complicated by the fact that an assignment through a pointer (or similar alias) must not kill previous assignments, because the pointer might not be pointing at the variable of interest. Because we do not know from a static analysis what a pointer points to, an assignment through a pointer that can point to V is a definition of V (and the store generated from it is a store into V); such definitions and stores are called *transparent*. In the presence of aliases, a sequence of definitions of a variable V and a sequence of stores into V might reach a breakpoint B along a given path p—this is treated in detail in Section 6. Only one definition (store) in the sequence is the last to assign into V, but with static analysis it is not known which one, because it is not known which pointers are aliases for V. A

[3]A store is an equivalence class by the same equivalence relation applied to definitions (having been determined by a compiler to represent the same or equivalent computations).

definition or store through a pointer *targets* V if the pointer is an alias for V at that definition or store. If some definition through a pointer targets V, the semantics of the program may require that some other definition through a pointer also targets V (e.g., if the value of the pointer has not changed). Also, we assume that on a given input, a pointer in the optimized version points to the same thing as the same pointer in the unoptimized version. This implies that if some definition through a pointer targets V, the semantics of the program may require that the store generated from the definition also targets V.

Definition 6. A transparent definition d of V is *turned off* if it is assumed not to target V, and d is *turned on* if it is assumed to target V. If d targets V, each definition or store that is thereby required by program semantics to target V is turned off if and only if d is turned off.

A symmetric treatment of *turned off* applies to stores, exchanging the roles of definitions and stores above.

The ith definition of V along a path is written d_i, and the ith store into V along a path is written s_i.

Definition 7. A store s_i *qualified-reaches a point Bk with definition* d_j *along a path-pair p to Bk* if d_j reaches Bk along p_u and if, when every d_t, $t > j$, that follows d_j on p_u is turned off (turning off stores generated from the d_t), and all other transparent definitions of V are turned on except those constrained to be off by some d_t (turning on corresponding stores); s_i is the store that assigns into V along p_o. A definition d_i *qualified-reaches a point Bk with store* s_j *along a path-pair p* similarly, exchanging the roles of stores and definitions in the previous sentence.

Definition 8. In the absence of assignments through aliases: a variable V is *current at a breakpoint B along path-pair p* iff the store into V that reaches B along p_o was generated from the definition of V that reaches B along p_u.

In the presence of assignments through aliases: V is *current at a breakpoint B along path-pair p* iff for all instances of definitions d_j and instances of stores s_i such that either s_i qualified-reaches B with d_j or d_j qualified-reaches B with s_i along p, s_i is generated from d_j.

Clearly, assignments through aliases are a crucial part of most procedural programming languages. Because of the complexity they introduce, a treatment of currency determination in their absence is given in Section 4, and throughout that section the simpler form of the definition can be assumed. The solution is then generalized in Section 6 to handle assignments through aliases.

Definition 9. *V is endangered at B along p* if it is not current at B along p.

Definition 10. In the absence of assignments through aliases: *V is noncurrent at B along p* iff no store into V that reaches B along p_o was generated from a definition of V that reaches B along p_u.

In the presence of assignments through aliases: V is *noncurrent at B along p* iff for all instances of definitions d_j and instances of stores s_i such that either s_i qualified-reaches B with d_j or d_j qualified-reaches B with s_i along p, s_i is not generated from d_j.

Definition 11. *V is current* at B iff V is current at B along each path-pair to B.

Definition 12. *V is endangered at B* iff it is endangered at B along at least one path-pair to B.

Definition 13. *V is noncurrent at B* iff V is noncurrent at B along each path-pair to B.

We turn now to how to determine which state of currency a variable is in at a breakpoint.

4. CURRENCY DETERMINATION

My approach to currency determination involves solving a set of a data flow equations. This requires control flow information and within-basic-block ordering information for definitions in the unoptimized program, and the same information for stores in the optimized program. It also requires a mapping between the unoptimized and optimized control flow information.

In this section, we simplify the presentation by assuming that no aliasing is present in the program. One consequence of this assumption is that only one definition and one store may reach a breakpoint along a single path. In Section 6 we present the mechanisms needed to allow aliasing.

Section 4.1 introduces *paired reaching sets*, which make up the data in the data flow computation. Section 4.2 discusses the data structure on which the computation is performed and its relationship to the unoptimized and optimized versions of a program. Section 4.4 discusses the aspects of the data flow computation that are particular to the problem of currency determination. Section 4.5 gives an algorithm for computing paired reaching sets at block boundaries. Section 4.6 extends it to compute paired reaching sets at breakpoints, and Section 4.7 describes how the results are used to determine the currency of a variable. Correctness proofs are given in Copperman [1993].

4.1 Paired Reaching Sets

I have introduced the terms "definition" and "store" to distinguish an assignment occurring in the source or unoptimized code from an assignment occurring in the optimized (or machine) code. I am now going to use definition/store pairs to convey some information about both a definition and the store generated from it.

Definition 14. A *ds-pair* is a pair (d, s), where d is a definition of a variable V, and s is a store into V.

If P is the ds-pair (x, y), $P.d$ is the definition element x, and $P.s$ is the store element y, giving the tautologies $(x, y).d = x$ and $(x, y).s = y$.

Because both definitions and stores are represented, the set of ds-pairs that reaches a breakpoint provides complete information about what should reach and what does reach the breakpoint. These sets are called **P***aired* **R***eaching* **S***ets*: PRS_B^V is the set of ds-pairs relevant to V that reach a breakpoint B. Loosely, $(d, s) \in \mathrm{PRS}_B^V$ means d is a definition of V that should reach B, and that s is a store into V that does reach B. More precisely, given a definition d of V and a store s into V, independent of whether s was generated from d:

—$(d, s) \in \mathrm{PRS}_B^V$ means there is a path-pair p such that d reaches B along p_u and s reaches B along p_o.

Given PRS_B^V:

—V is current at B iff $\forall (d, s) \in \mathrm{PRS}_B^V$, s was generated from d;

—V is endangered at B iff $\exists (d, s) \in \mathrm{PRS}_B^V$ such that s was not generated from d;

—V is noncurrent at B iff $\forall (d, s) \in \mathrm{PRS}_B^V$, s was not generated from d.

4.1.1 *Caveat.* An infeasible path in a flow graph is one that cannot be taken in any execution, and a feasible path is one that can be taken in some execution. Infeasible paths introduce conservative error under this currency determination technique. The claims just made hold for programs without infeasible paths. If (d, s) is in PRS_B^V by virtue of an infeasible path, and s was not generated from d, this technique will claim that V is endangered though it may be current. However, we can do no better than the compiler, and like the compiler, we must make the conservative assumption that all paths are feasible.

4.2 The Flow Graph Data Structure

The relationship between the optimized and unoptimized code must be captured in a data structure that can be used by the currency determination algorithm. We assume that the information used to do currency determination on a compilation unit is produced in a single compilation. Also, information about the unoptimized version of the program is taken from the compiler's intermediate representation of the program prior to the optimizing phases (independent of whether code is generated for an unoptimized version); thus the full facilities of the compiler are available to produce information about the relationship between the unoptimized "version" and the optimized version.

Currency determination needs the following:

—the assignments that constitute a definition,

—the "generated from" relationship between definitions and stores (from these two pieces of information, the machine code instructions that constitute a store can be determined),

—the execution order of statements and side effects within a basic block, for blocks in both the optimized and unoptimized versions, and

—the correspondence between the flow graphs for each version.

The particular encoding of the information is not important here. One possible encoding is described in Appendix A. Here we assume that the first three items are available and discuss the fourth.

Some data structures must represent the flow graphs for each version and the correspondence between them. Hereafter, the term *source graph* refers to the flow graph for the unoptimized version, and *object graph* refers to the flow graph for the optimized version. *DS-graph* refers to the data structure used to map between them, upon which the data flow computation is performed.

The DS-graph construction is constrained such that a node in the DS-graph is derived from a block in the source graph, from a block in the object graph, or from both. The constraints are described in Section 4.3. A node B in the DS-graph *selects* the block(s) it is derived from: B selects at most one basic block B_u in the source graph, and B selects at most one basic block B_o in the object graph. B contains ordering information about definitions occurring in B_u in a definition list. If no block B_u is selected, B contains an empty definition list. Similarly, B contains ordering information about stores occurring in B_o in a store list—if no block B_o is selected, B contains an empty store list. A path p through the DS-graph selects p_u where p_u is the sequence of blocks in the source graph selected by the sequence of nodes in p, and p selects sequence p_o in the object graph similarly. By DS-graph construction, p_u forms a path through the source graph, and p_o forms a path through the object graph; and $\langle p_u, p_o \rangle$ is a path-pair.

The correspondence between blocks and edges in the source and object graph is immediate when optimization has not caused the object graph to differ in shape from the source graph. When optimization causes changes in the object graph, these changes must be reflected in the DS-graph.

If $(d, s) \in \text{PRS}_B^V$ is to mean that there is a path-pair $\langle p_u, p_o \rangle$ such that d reaches B along p_u and s reaches B along p_o, then the path through the DS-graph that caused (d, s) to be placed into PRS_B^V must select $\langle p_u, p_o \rangle$.

A feasible path in the DS-graph is one that selects paths that can be taken in some execution. The DS-graph will contain infeasible paths (i.e., paths that select path-pairs through which execution cannot proceed) if the object graph contains infeasible paths. It would clearly be preferable to construct the DS-graph so that it contains only feasible paths, but it is not possible in general to determine which paths through a flow graph are feasible.

Definition 15. A DS-graph is *valid* if and only if every path p through the DS-graph denotes a path-pair, that is, the pair of paths $\langle p_u, p_o \rangle$ selected by p is a path pair, and every feasible path-pair is denoted by some path through the DS-graph.

For the purposes of this discussion, we assume that for each node in the DS-graph,

—There is a list of the definitions that are in the block in the source graph selected by the node, in execution order. Because this list contains definitions, I call it the *definition list*. This is equivalent to a list of statements and side effects that appear in that block, in the order in which they appear in the source code.

—There is a list of the stores that are in the block in the object graph selected by the node, in execution order. Because this list contains stores, I call it the *store list*. This is again equivalent to a list of statements and side effects that appear in that block, but execution order in the optimized version is not equivalent to source order.

The definition list must order definitions relative to possible breakpoint locations. Similarly, the store list must order stores relative to possible breakpoint locations.

These assumptions guarantee that if a node appears in a path through a valid DS-graph, the code in the unoptimized version represented by that node is the code that would be executed (in proper order) if the block represented by that node was visited on that path, and we have the analogous guarantee relative to the optimized version.

A node n in the DS-graph may select a block in one version that does not exist in the other version, in which case n does not select any block in the latter version. The DS-graph is constructed so that the appropriate list (definition or store) is empty for n. For example, if a loop preheader was introduced by optimization, the DS-graph will have a node that (1) selects the preheader in the object graph, (2) does not select any block in the source graph, and (3) has an empty definition list.

4.3 Graph Transformations

The DS-graph begins isomorphic to the source graph, with definition lists replacing the code within blocks, and where each node selects the block it maps to. Before any optimizations change its shape, it is clearly valid.

DS-Graph Creation Rule. A DS-graph is created before any optimization has been performed. At the same time, the object graph is created. The object graph is a copy of the source graph. For each block B_u in the source graph, a DS-graph node B is created such that B selects B_u and B_o. The definition list for B is abstracted from the code in B_u and copied to the store list for B. For each edge (h_u, t_u) in the source graph, edge (h_o, t_o) is in the object graph, and edge (h, t) is in the DS-graph. The name of a block in the source graph is subscripted by u, and the name of a block in the object graph is subscripted by o; thus B_o is a copy of B_u.

Any changes made to the shape of the flow graph by optimization are reflected in changes to the shape of the DS-graph and are constrained by the graph transformations enumerated below, pictured in Figures 6 to 11, and formally described in Appendix B and in Copperman [1993]. These transformations maintain the validity of the DS-graph. The currency determination method applies only to optimizations that change the flow graph in a manner that can be modeled by iterative application of these transformations. A transformation on the DS-graph accompanies each transformation on the object graph; thus a transformation is applied only if it is semantically valid.

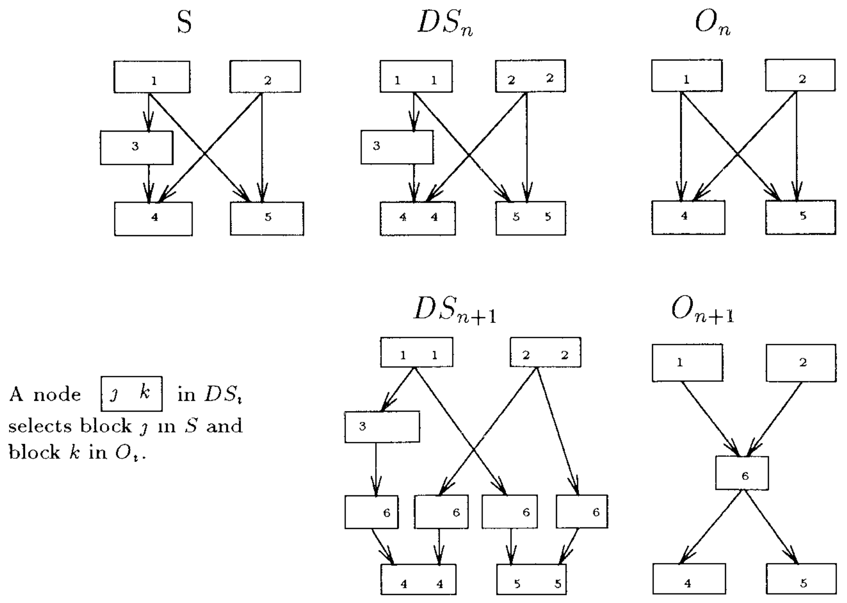

S
DS_n
O_n
DS_{n+1}
O_{n+1}
A node $\boxed{j \quad k}$ in DS_i selects block j in S and block k in O_i.

Fig. 6. Graph Transformation 1: Introducing a block. In the object graph, a block is introduced between a complete bipartite subgraph. In DS_n, there is a path corresponding to each edge deleted in the object graph transformation. A node selecting the introduced block is added to the end of each such path

Some of the graph transformations copy nodes in the DS-graph when at first glance it seems unnecessary. The reason for these copies is that when transformations are composed, operations are done on all subpaths in the DS-graph that correspond to an edge in the object graph. If a node in the DS-graph were on more than one such subpath, the transformations could not successfully be composed.

Graph Transformations.

(1) Introducing a block. An example is shown in Figure 6.
(2) Deleting a block (shown in Figure 7).
(3) Deleting an edge (Figure 8).
(4) Coalescing two blocks into a single block (Figure 9).
(5) Inlining a subroutine (Figure 10).
(6) Unrolling a loop (Figure 11).
(7) No optimization other than those described in one of the other object graph transformations may modify control flow in a way that changes the order that blocks are entered on a particular input. A mechanism that allows truthful (but not expected) behavior in the presence of optimizations that violate this constraint is described in Copperman [1993].

Given a valid DS-graph, if any of these modifications are performed, the result is a valid DS-graph. Note that optimizations that do not modify the

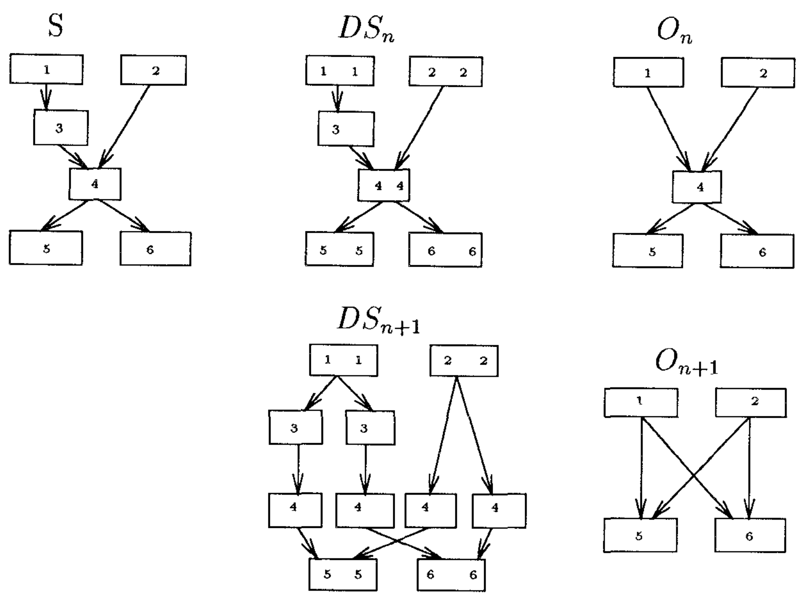


Fig. 7. Graph Transformation 2: Deleting a block. In the object graph, predecessors and successors of the deleted block form a complete bipartite subgraph in the resulting graph. In the DS-graph, one path is constructed for each edge introduced in the object graph transformation. Nodes are duplicated so that each such path selects the requisite blocks in the source graph.

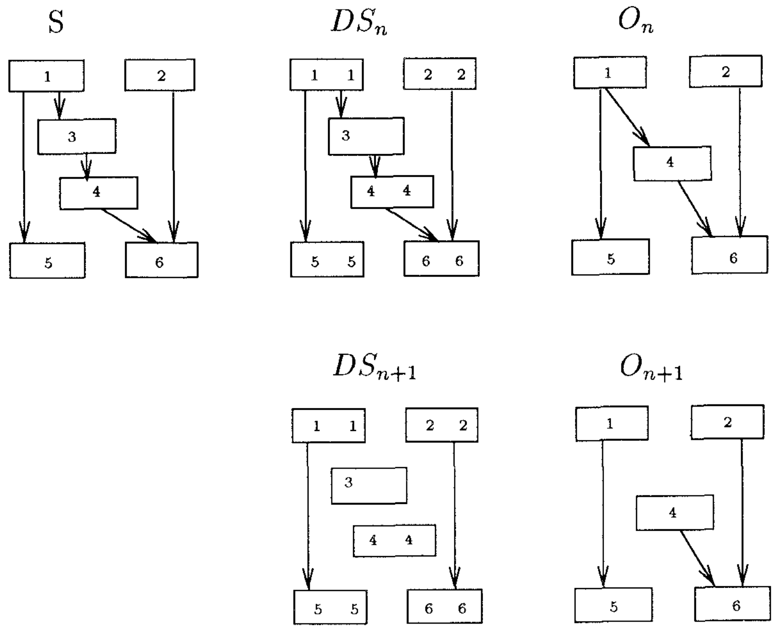


Fig. 8. Graph Transformation 3: Deleting an edge. An edge is deleted in the object graph. Paths in DS_n selecting that edge are deleted, and all edges out of nodes in DS_n that are subsequently unreachable are deleted.

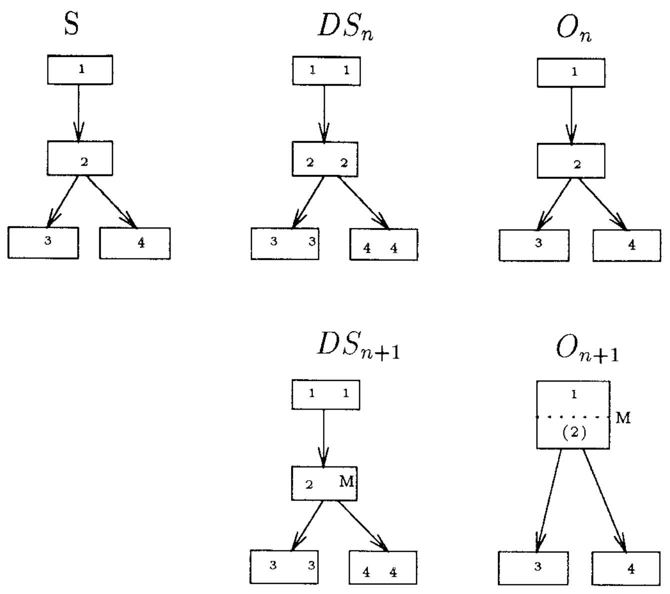


Fig. 9. Graph Transformation 4: Coalescing two blocks. A block boundary marker is placed in the object graph resulting from the coalescing, and that marker is referenced by the resulting DS graph.

shape of the flow graph preserve the validity of the DS-graph. Many optimizations modify the flow graph only in ways that can be modeled by these DS-graph modifications, but arbitrary optimization is not modeled. For example, recognizing bubblesort in the unoptimized version and replacing it with quicksort in the optimized version clearly involves graph modifications beyond the scope of these transformations. Loop interchange cannot be modeled because it changes the order in which blocks are entered.

The data flow computation is performed on the DS-graph. We turn now to the nature of that computation.

4.4 Dataflow on DS-Pairs

4.4.1 *The Gen DS-Pair.* Analogous to the Gen set of standard data flow algorithms, a Gen ds-pair contains information about what is generated in a block. In the absence of aliasing, there is exactly one Gen ds-pair for a given variable in each block. The Gen ds-pair for a variable V and a block B is written Gen_B^V. Gen_B^V is $\langle d, s \rangle$, where d is the last definition of V on B's definition list or *null* if there is none, and s is the last store into V on B's store list or *null* if there is none. Only Gen ds-pairs may have null entries. Ds-pairs with null entries do not appear in the In or Out sets of a block or in PRS_B^V.

4.4.2 *The κ Operation.* In a standard data flow computation, a definition in a block kills definitions (of the same variable) that reach the entry of the block. In the currency determination data flow computation given in Algo-

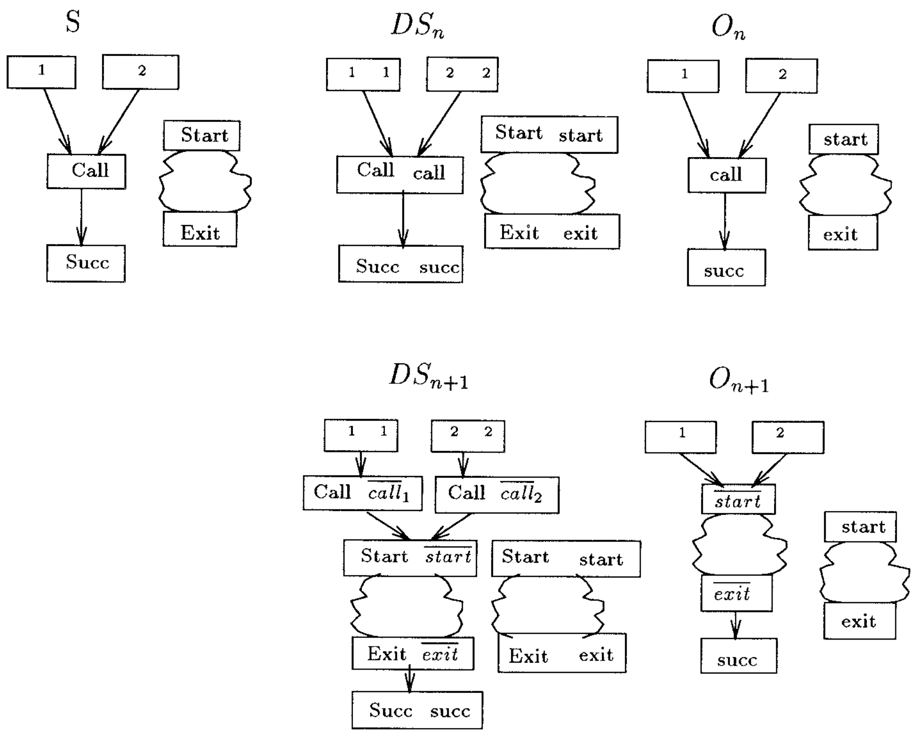


Fig. 10. Graph Transformation 5: Inlining a subroutine. The subroutine is copied into the DS-graph. The duplication of the call node is necessary to preserve DS-graph characteristics need by other transformations.

rithm PRS, definitions kill definitions, and stores kill stores. If $\text{Gen}_B^V.\text{d} = d$, ds-pairs reaching the exit of B contain d as their definition element. If $\text{Gen}_B^V.\text{d} = null$, ds-pairs reaching the exit of B contain the same definition element they had at the entry of B. Stores are handled analogously. This is represented with the operator κ.

Definition 16.

$$(e,t)\kappa(null, null) = (e,t)$$
$$(e,t)\kappa(d, null) = (d,t)$$
$$(e,t)\kappa(null, s) = (e,s)$$
$$(e,t)\kappa(d,s) = (d,s)$$

4.4.3 *The* $\overset{\kappa}{\bigcup}$ *Operation.* Multiple ds-pairs may reach the entry of a block. The $\overset{\kappa}{\bigcup}$ operation defines the ds-pairs that reach the exit of a block, given all of the pairs that reach block entry. The $\overset{\kappa}{\bigcup}$ operation takes a set (the In set for a block) as its left operand, whereas the κ operation takes a single ds-pair as its left operand. If the block generates both a store and a definition, its Out

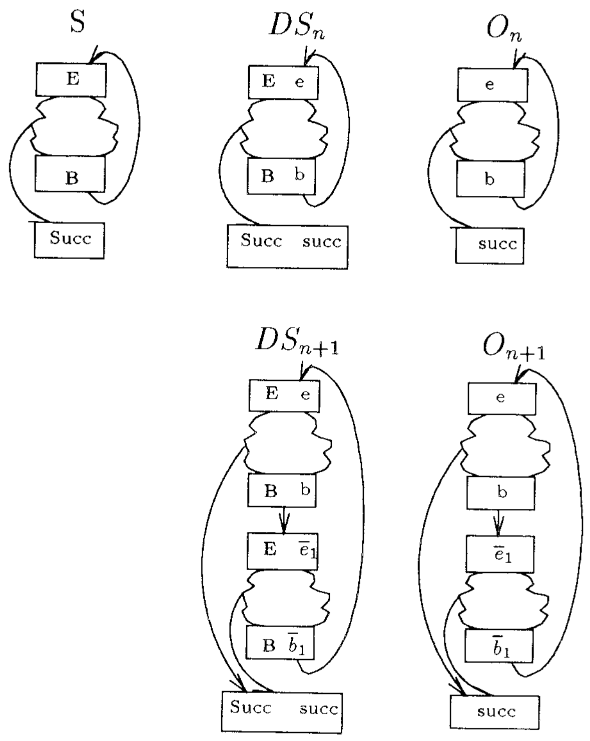


Fig. 11. Graph Transformation 6: Unrolling a loop. The body of the loop is unrolled in the DS-graph in the same manner that it is unrolled in the object graph. The unshown body of the loop may differ between the source graph, object graph, and DS-graph due to previous transformations, but varies between O_n and O_{n+1}, only in the manner shown.

set will contain the ds-pair consisting of that store and definition, and if it contains a null in either position (or both), its Out set depends on the In set. Since every variable is defined to have an initial definition and store, propagation will eventually cause any reachable block to have a nonempty In set.

If the In set is empty, the Out set is the set of complete ds-pairs (those containing no *nulls*) in the Gen set, and if the In set is not empty, the Out set is the set of ds-pairs produced by individual κ operations between each ds-pair in R and the ds-pair S:

Definition 17.

$$Complete(S) = \begin{cases} \varnothing & \text{if } d = null \text{ or } s = null \\ (d, s) & \text{otherwise} \end{cases}$$

Definition 18.

$$R \overset{\kappa}{\bigcup} S = \begin{cases} Complete(S) & \text{if } R = \varnothing \\ \{r \kappa S \mid r \in R\} & \text{otherwise} \end{cases}$$

```
Initialize
Input:
   a component of the DS-graph, modified by the addition of a Start node;
Output:
   the Gen sets of each variable for each block.

   for each variable $V$
      $\mathrm{Gen}^V_{Start} = (d\text{-}init, s\text{-}init)$
      for each node $B$ other than $Start$
         set $\mathrm{Gen}^V_B.d$ to the last definition of $V$ in the definition list of $B$
            or to $null$ if there is none
         set $\mathrm{Gen}^V_B.s$ to the last store into $V$ in the store list of $B$
            or to $null$ if there is none
End of Initialize

Iterate
Input:
   a component of the DS-graph, modified by the addition of a Start node,
   the Gen sets of each variable for each block;
Output:
   the paired reaching sets of each variable at each block boundary

   for each variable $V$
      for each block $B$
         $\mathrm{In}^V_B = \mathrm{Out}^V_B = \emptyset$
      iteratively compute $\mathrm{In}^V_B$ and $\mathrm{Out}^V_B$ until convergence, according to the following,
         for each block $B$
            $\mathrm{In}^V_B = \bigcup_P \mathrm{Out}^V_P$ for $P$ predecessors of $B$
         for each block $B$
            $\mathrm{Out}^V_B = \mathrm{In}^V_B \mathbin{\hat{\cup}} \mathrm{Gen}^V_B$
End of Iterate

Algorithm PRS
Input:
   a component of the DS-graph, modified by the addition of a Start node,
Output:
   the paired reaching sets of each variable at each block boundary.

   Initialize
   Iterate
End of Algorithm PRS
```

Fig. 12. Algorithm PRS computes paired reaching sets at block boundaries for a DS-graph component (a subroutine). The algorithm consists of an initialization step **Initialize** followed by an iterative step **Iterate**.

4.5 Paired Reaching Sets at Block Boundaries

Algorithm PRS, given in Figure 12, computes paired reaching sets at block boundaries for a DS-graph component (a subroutine). A source node, *Start*, is grafted on to the component to provide a place for initial definitions (*d-init*) and stores (*s-init*) representing the creation of variables. The algorithm

```
Initialize-BK
Input
  a variable $V$, a syntactic breakpoint $Bk$, and the node $B$ containing $Bk$.
Output:
  The Gen set of $V$ for $B$ at $Bk$,
  Set $\mathrm{Gen}^V_{Bk}.d$ to the last definition of $V$ prior to $Bk$ on $B$'s definition list,
      or to null if there is none
  Set $\mathrm{Gen}^V_{Bk}.s$ to the last store into $V$ prior to $Bk$ on $B$'s store list,
      or to null if there is none
End of Initialize-BK

Algorithm PRS-BK
Input
  a variable $V$, a syntactic breakpoint $Bk$, the node $B$ containing $Bk$, and $\mathrm{In}^V_B$,
Output
  The paired reaching set of $V$ at $Bk$;
  Initialize-BK
  $\mathrm{PRS}^V_{Bk} = \mathrm{In}^V_B \hat{\cup} \mathrm{Gen}^V_{Bk}$
End of Algorithm PRS-BK
```

Fig. 13 Algorithm PRS-BK computes the paired reaching set for a variable V at a syntactic breakpoint within a block B given the paired reaching set for V at B's entry. Initialize-BK computes the Gen set for the breakpoint, which modifies the In set to produce the desired result. No iteration is needed.

consists of an initialization step and an iterative step. (If the DS-graph component is a subroutine, line 1 of Initialize should set Gen^V_{Start} to (*d-incoming*, *s-incoming*) when V is a parameter.)

4.6 Paired Reaching Sets at Breakpoints

Algorithm PRS provides In and Out sets at block boundaries. Our goal is to determine a variable's currency at a breakpoint, which may be in the middle of a block rather than at a block boundary. The definitions of current, noncurrent, and endangered refer to "a breakpoint B" that lies on a path-pair. This is well defined for syntactic breakpoints, but not for semantic breakpoints whose representative instruction has been moved by optimization, so we compute paired reaching sets at syntactic breakpoints only. The consequences of this are that a block containing the breakpoint is present in both the source and the object graphs, and the breakpoint is on both the definition list and the store list for some block in the DS-graph. Note that for a syntactic breakpoint for a statement that has been moved or eliminated, the element on the store list representing the breakpoint is the representative instruction for a following statement.

PRS^V_{Bk}, the set of ds-pairs relevant to V that reach a breakpoing *Bk*, is derived from the In sets computed by Algorithm PRS by Algorithm PRS-BK, given in Figure 13.

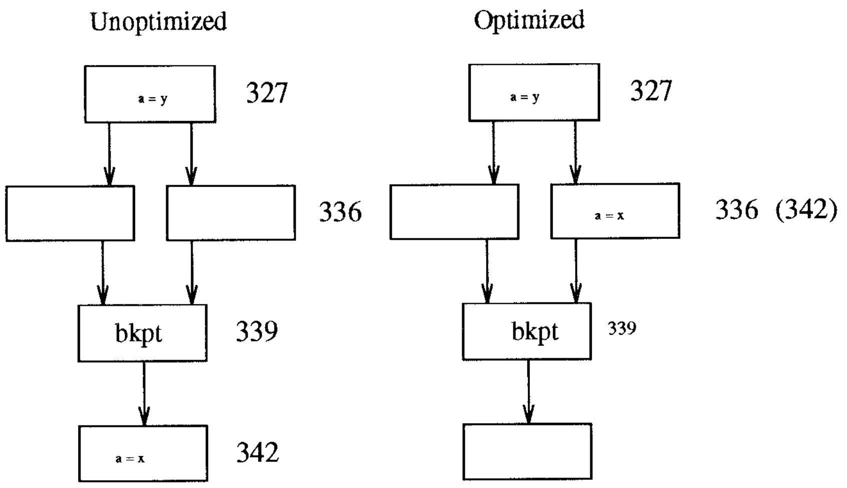


Fig. 14. The display of a could be accompanied by this message: "Breakpoint 1 has been reached at line 339. a should have been set at line 327. However, optimization has moved the assignment to a at line 342 to near line 336. a was actually set by one of lines 327 or 342.

4.7 A Variable's Currency

The contents of PRS_{Bk}^{V} tell us V's currency.

THEOREM 19. $\text{PRS}_{Bk}^{V} = \emptyset$ *iff either V is not in scope at Bk or Bk is unreachable. Otherwise*:

—*V is current at Bk iff* $\forall (d, s) \in PRS_{Bk}^{V}$, *s was generated from d*;

—*V is endangered at Bk iff* $\exists (d, s) \in PRS_{Bk}^{V}$ *such that s was not generated from d*;

—*V is noncurrent at Bk iff* $\nexists\, (d, s) \in PRS_{Bk}^{V}$ *such that s was generated from d*.

Theorem 19 is proven in Copperman [1993].

5. WHEN A VARIABLE IS ENDANGERED

When the debugger is asked to display a variable, it determines whether the variable is current. If the variable is current, the debugger displays its value without comment. If the variable is endangered, in addition to displaying its value, the debugger can give the user some help in understanding why the value is endangered. The general flavor of what the debugger can do is given by the following sample message that might accompany the display of variable a when the optimization shown in Figure 14 has occurred.

Breakpoint 1 has been reached at line 339; a should have been set at line 327. However, optimization has moved the assignment to a at line 342 to near line 336; a was actually set by one of lines 327 or 342.

The information contained in this message is available from the paired reaching set PRS^a_{339} and the unoptimized and optimized flow graphs. The description of the effects of optimization will vary in specificity as the effects of optimization vary in complexity.

6. TRANSPARENCY

6.1 Assignments Through Aliases

Consider an assignment $*P$ through a pointer (or through an array element where the index is a variable) that could point to V. When execution is suspended at a breakpoint B, $*P$ may be an alias for V. $*P$ must be considered to be a definition of V that reaches B. If $*P$ is not an alias for V in some particular execution, the value that V contains at the breakpoint came from whatever definition would have reached if $*P$ were not present. Therefore, this definition must also be considered to reach B. For any language that allows such aliasing, the assumption of a single definition reaching along a given path-pair does not hold.

If there are multiple definitions of V that reach B along p, all of them but one (the one furthest from B on p) must be assignments through aliases, because other kinds of assignments kill prior definitions. An assignment through an alias is defined as such by its ambiguity about whether V is assigned into, because if it can be determined that an assignment through a pointer does assign into V every time, that assignment kills prior definitions, and if it can be determined that an assignment through a pointer never assigns into V, the assignment is not a definition of V.

6.2 Definitions and Stores Revisited

The computation of paired reaching sets depends on the definition of κ given in Section 4.4, which in turn depends on the assumption that a data object assigned a value within a basic block is unconditionally affected by the assignment. This plays out as an assumption that definitions kill definitions and stores kill stores. This assumption holds for direct assignments but not for assignments through aliases. Because we have two kinds of assignments with different characteristics, we need two kinds of definitions and stores. Definitions and stores that are unambiguous as to the variable that is affected we call *opaque*. Definitions and stores that affect one of a set of variables such that it cannot be determined at compile time which variable will be affected, we call *transparent*. Thus `*p = 0;` is a transparent definition of V unless the compiler determines that `p` cannot point to V at that assignment. V is one member of the set of variables that might be affected by the definition, and `p` is called a *transparent definer*.

It may be that `p` never points to V, but the compiler cannot determine that. The debugger can do no better; we must treat assignments through `p` as if they can affect V.

It can be shown that a compiler can make an assignment through a pointer that is not current without violating program semantics [Copperman 1993]. The circumstances under which a compiler can do so are sufficiently con-

Table I. This Redefinition of κ Overgenerates ds-Pairs

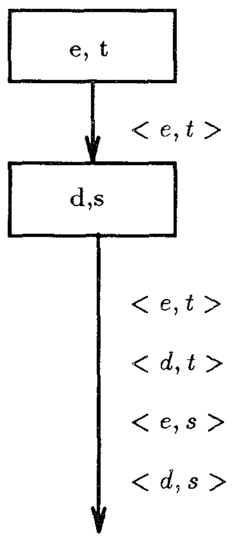
$(e,t)\,\kappa\,(d,s)$	d is null	d is opaque	d is transparent
s is null	(e,t)	(d,t)	(e,t) (d,t)
s is opaque	(e,s)	(d,s)	(e,s) (d,s)
s is transparent	(e,t) (e,s)	(d,t) (d,s)	(e,t) (d,s) (e,s) (d,t)

strained that, for the remainder of this work, I assume that transparent definers are current at each of their uses.

6.3 κ Revisited

The introduction of transparent definitions requires a redefinition of the κ operator so that transparent definitions and stores do not kill elements that should reach subsequent points in the program. The right operand of κ is doing the killing, so we do not need to worry about the transparency of the left operand. We could redefine κ by independently letting an opaque definition of V kill other definitions of V, transparent and opaque alike, and letting an opaque store into V kill stores into V, transparent and opaque alike, while not letting a transparent definition of V kill definitions of V and not letting a transparent store into V kill stores into V. Such a definition is given in Table I. However, this overgenerates ds-pairs.

As a motivating example, suppose there is a block B containing the definition d whose source code is `*p = x`, where `p` may point to V, and the store s generated from d has survived the optimizer without being moved or eliminated. Gen_B^V is (d, s), and d and s are transparent. Assume that no optimization has affected assignments into `p`. Then on a given execution, either V is affected (`p` points to V) or V is not affected (`p` does not point to V). In this case, although $(d, t) \in (e, t)\kappa(d, s)$, there is no input on which d reaches the exit of B in the unoptimized version, and t reaches the exit of B in the optimized version. This is illustrated in Figure 15.



```
p = &V;

...

if (cond) {

   p = &W;

   }

...

*p = x; // This is d.
```

Fig. 15. An Infeasible Ds-pair: e is a definition of V, and t is a store into V. Whether d is a definition of V and s is a store into V depends on which assignment to p reaches the assignment through p. (d, t) is an infeasible ds-pair because d defines V only when s stores into V, in which case s kills t. (e, s) is an infeasible ds-pair for similar reasons.

In general, let s be a store through some pointer, d be the definition that generated s, and B be a breakpoint. If in every execution in which s can affect V at B, d reaches B, then at B, any ds-pair (d, x) where $x \neq s$ or (x, s) where $x \neq d$ is called *infeasible*. Such a ds-pair does not represent a definition that reaches B in the unoptimized code and the store that reaches B on the same input in the optimized code. The definition of κ in Table I places infeasible ds-pairs in paired reaching sets, and its use would result in conservative errors in the results.

6.4 Constrained Transparency

Many compilers do not do sufficient pointer analysis to optimize transparent assignments, and therefore do not eliminate or move them. Handling transparency in its full generality is more complex and costly than currency determination in the absence of transparency. The constraints that a lack of pointer analysis imposes on the optimizer allow for a middle ground in terms of cost and complexity. Copperman [1993] describes algorithms that take advantage of these constraints.

6.5 Unconstrained Transparency

If transparent assignments may be eliminated or moved, a transparent ds-pair in a Gen set may contain nulls. Furthermore, a block may contain a definition d and a store s not generated from d. As a consequence, there are

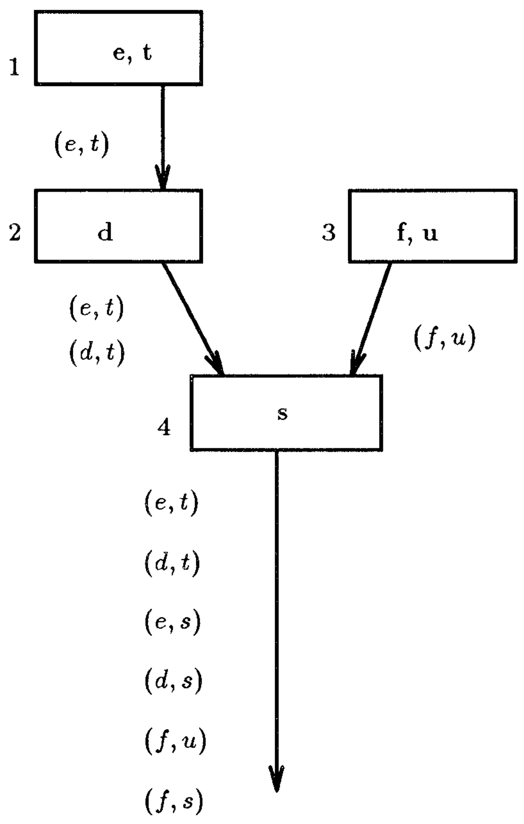


Fig. 16. If transparent store s is not generated from transparent definition d, all ds-pairs in the Out set of node 4 are feasible. If s is generated from d, (d, t) and (e, s) are infeasible. However, (f, s) is feasible. The salient difference between (e, s) and (f, s) is that e and d reach node 4 along the same path, while f and d do not.

circumstances in which any of the ds-pairs produced by the definition of κ given in Table I are feasible and other circumstances in which some of them are infeasible. Figure 16 exemplifies the impossibility of producing only feasible ds-pairs in the presence of unconstrained transparency with the mechanisms presented so far. It is possible to distinguish the circumstances in which κ will produce an infeasible ds-pair from the circumstances in which it will produce a feasible ds-pair, but only with knowledge of which definitions or stores are generated along the same path—information not available to κ.

To capture this further information, ds-pairs are extended to ds-list-pairs—pairs of lists rather than pairs of elements. The definition element in a ds-list-pair for a variable V is the list of definitions of V that reach a block B along a path p, in the order that they occur on p. The store element is the list of stores into V that reach B along p, similarly ordered. It follows that the first element in each list is opaque, and the rest are transparent. Figure 17 is the example from Figure 16 using ds-list-pairs instead of ds-pairs.

The type of κ's operands has been modified, so it is necessary to redefine κ. The list operations are the same for definitions and stores, and are performed independently on definitions and stores, so I introduce a list operator ℓ. The

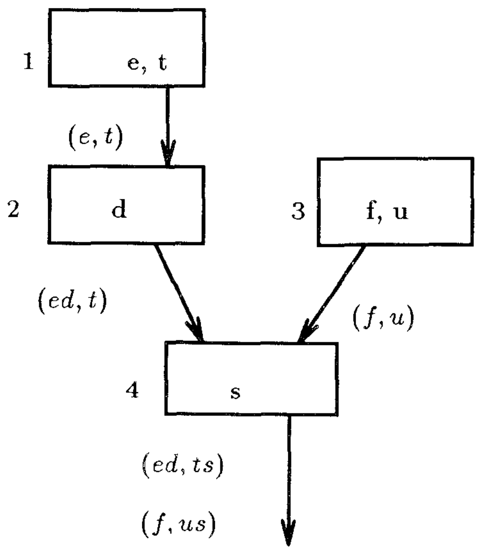


Fig. 17. This is the example from Figure 16 recast to use ds-list-pairs. Encoded in the ds-list-pairs is the fact that e and d reach node 4 along the same path while f and d do not.

empty list is expressed as *null*, and | is used as a list concatenation operator. Given two lists x and y, we have the following definition.

Definition 20.

$$x \not\ell y = \begin{cases} x & \text{if } y = null \\ y & \text{if an element of } y \text{ is opaque} \\ x|y & \text{if all elements of } y \text{ are transparent} \end{cases}$$

Then given two ds-list-pairs (e_l, t_l) and (d_l, s_l):

Definition 21. $(e_l, t_l)\kappa(d_l, s_l) = (e_l \not\ell d_l, t_l \not\ell s_l)$.

With multiple assignments encoded in a single ds-list-pair, we return to a situation in which the Gen set for a block is a single entity.

6.5.1 $\overset{\kappa}{\bigcup}$ *in the Presence of Unconstrained Transparency.* *Complete* must be redefined to act on ds-list-pairs:

Definition 22.

$$Complete((d_l, s_l)) = \begin{cases} \varnothing & \text{if } d_l \text{ or } s_l \text{ is } null \\ (d_l, s_l) & \text{otherwise} \end{cases}$$

Definition 21 defines κ as an append operation on transparent definitions. Repeated κ operations on a loop containing only transparent definitions of V result in lists that grow indefinitely, since the definitions and stores within the loop get appended ad infinitum. We can construct ds-list-pairs that do not grow indefinitely by including only the last occurrence of any duplicated definitions and stores. There are cases in which information about currency is lost unless the last two instances of a definition or store are included in the ds-list-pairs. In these cases, the choice is between an algorithm that has the potential for nonconservative error (in which an endangered variable may be reported as current) and a more complex algorithm that has greater potential for conservative error (a current variable may be reported as endangered). These cases are discussed in Section 6.7. For reasons presented in that section, I have chosen the former of these alternatives; thus the algorithm presented below has the potential for nonconservative error.

The ds-list-pair construction method given below preserves the characteristic that the ith store in a ds-list-pair is generated from the ith definition in that ds-list-pair for all i if and only if V is current along the path from which that ds-list-pair is derived, except in the cases discussed in Section 6.7 (this is shown in Copperman [1993]).

Definition 23. Let $x_l = \langle x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_n \rangle$ where x_i and x_j, are in the same equivalence class. Then $last(x_l) = \langle x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_j, \ldots, x_n \rangle$, and $last^*(x_l)$ is the result of applying $last$ to x_l repeatedly until each equivalence class is represented at most once. Also, $Last(d_l, s_l) = (last^*(d_l), last^*(s_l))$.

$$R \overset{\kappa}{\bigcup} S = \begin{cases} Last(Complete(S)) & \text{if } R = \emptyset \\ Last(Complete\ (S)) & \text{if } \exists e \in Complete(S) \text{ such that } e \text{ is opaque} \\ \{Last(r\kappa s) \mid r \in R,\ s \in S\} & \text{otherwise} \end{cases}$$

6.6 Currency in the Presence of Unconstrained Transparency

Figures 18 and 19 present algorithms modified to handle unconstrained transparency. Again, most of the additional work has been incorporated into the definition of $\overset{\kappa}{\bigcup}$, so only the initialization phase of each algorithm needs modification.

Once again, the contents of PRS^V_{Bk} tell us V's currency. Excepting the cases discussed in Section 6.7:

—The ith store in a ds-list-pair is generated from the ith definition in that ds-list-pair for all i if and only if V is current along the path from which that ds-list-pair is derived.

—Each ds-list-pair e tells us whether V is current along the set of paths from which e is derived.

—All paths are represented by some ds-list-pair in PRS^V_{Bk}.

In other words, PRS^V_{Bk} contains the interesting ds-list-pairs, that is, PRS^V_{Bk} contains a set of ds-list-pairs that is (nearly) sufficient to determine V's currency.

Definition 24. *V is current at Bk by ds-list-pair e* iff V is current along each path p to Bk such that e is derived from p.

The following theorem does not hold in the cases discussed in Section 6.7.

THEOREM 25. *$PRS^V_{Bk} = \emptyset$ iff either V is not in scope at B or B is unreachable. Otherwise, V is current at Bk iff V is current at Bk by e, $\forall e \in PRS^V_{Bk}$; V*

```
Initialize$^T$
Input:
        a component of the DS-graph, modified by the addition of a $Start$ node;
Output:
        the Gen sets of each variable for each block.

        for each variable $V$
            $\mathrm{Gen}^V_{Start} = (d\text{-}init, s\text{-}init)$
            for each node $B$ other than $Start$
                $d_l = null$
                for each definition $d$ of $V$ in $B$'s definition list, in order of appearance
                    $d_l = d_l \ell d$
                $s_l = null$
                for each store $s$ into $V$ in $B$'s store list, in order of appearance
                    $s_l = s_l \ell s$
                $\mathrm{Gen}^V_B = (d_l, s_l)$
End of Initialize$^T$

Algorithm PRS$^T$
Input:
        a component of the DS-graph, modified by the addition of a $Start$ node,
Output
        the paired reaching sets of each variable at each block boundary.

        Initialize$^T$
        Iterate
End of Algorithm PRS
```

Fig. 18. Algorithm PRS^T computes paired reaching sets at block boundaries in the presence of transparency. The initialization step is modified, but the iterative step differs only in the definition $\bigcup^{K}$; so the previous version of Iterate can be used.

is endangered at Bk iff $\exists e \in PRS^V_{Bk}$ *such that V is not current at Bk by e*; *V is noncurrent at Bk iff* $\nexists\, e \in PRS^V_{Bk}$ *such that V is current at Bk by e.*

In Copperman [1993], Theorem 25 is proven for most cases. It is shown how it fails for some exceptional cases, which are discussed below.

6.7 Cases in which the Algorithm May Err

Consider the case in which a loop-invariant transparent assignment to V is moved to a loop preheader. Let the loop be L, the invariant definition of V in L be dx, and the store generated from dx be sx. (In general, I use the naming convention that a definition dx generates the store sx.) Let p be a path to a breakpoint Bk (where Bk is after L) that contains two or more iterations of L. Although the argument made here does not depend on it, for simplicity of presentation assume that no other transparent assignments to V reach Bk. Then the sequence of definitions of V that reach Bk along p is $\langle d_1, d_2, d_3, \ldots, d_n \rangle$ where there are $n-1$ iterations of L in p, and d_2 through d_n are instances of dx. The sequence of stores into V that reach Bk

```
Initialize-BK$^T$
Input
            a variable $V$, a syntactic breakpoint $Bk$, and the node $B$ containing $Bk$;
Output:
            The Gen set of $V$ for $B$ at $Bk$;
            $d_l = null$
            for each definition $d$ of $V$ prior to $Bk$ on $B$'s definition list, in order of appearance
                $d_l = d_l \ell d$
            $s_l = null$
            for each store $s$ into $V$ prior to $Bk$ on $B$'s store list, in order of appearance
                $s_l = s_l \ell s$
            $\mathrm{Gen}^V_{Bk} = (d_l, s_l)$
End of Initialize-BK$^T$

Algorithm PRS-BK$^T$
Input:
            a variable $V$, a syntactic breakpoint $Bk$, the node $B$ containing $Bk$, and $\mathrm{In}^V_B$,
Output:
            The paired reaching sets of $V$ at $Bk$;
            Initialize-BK$^T$
            $\mathrm{PRS}^V_{Bk} = \mathrm{In}^V_B \hat{\cup} \mathrm{Gen}^V_{Bk}$
End of Algorithm PRS-BK$^T$
```

Fig. 19. Algorithm PRS-BK^T computes paired reaching sets at syntactic breakpoint in the presence of transparency in a similar manner to Algorithm PRS-BK; again the initialization step is modified.

along p is $\langle s_1, s_2 \rangle$ where s_2 is sx, in the preheader. d_1 and s_1 are the opaque definition of V and store into V that reach Bk from above the loop.

Assuming that s_1 is generated from d_1, V is current at Bk: either s_1 qualified-reaches Bk with d_1, or s_2 qualified-reaches Bk with d_n. (s_2 cannot qualified-reach with d_i, $1 < i < n$, because that would turn off d_n, which would turn off not only s_2 but also d_i.) This case is handled correctly—ds-list-pair $(\langle d_1, dx \rangle, \langle s_1, sx \rangle)$ is derived from p. This same ds-list-pair is derived no matter how many times p traverses L.

Note that if the last two duplicate definitions in a loop were recorded in ds-list-pairs, then we would have the ds-list-pair above from a path that traverses L once; and we would have ds-list-pair $(\langle d_1, dx, dx \rangle, \langle s_1, sx \rangle)$ from paths that traverse L more than once; and we would report that V is endangered, because of the mismatching ds-list-pair.

Now suppose instead that the left-hand-side of dx and sx is not loop invariant. Turning off some d_i, $i > 1$, does not turn off all d_i, so s_1 qualified-reaches Bk with any d_i. In this case, V is endangered, but would be reported as current. Note that if the last two duplicate definitions in a loop were recorded in ds-list-pairs, then we would report V as endangered. This is a semantically pathological case, since the compiler could only move sx to a preheader if it determined that all data that sx can affect is dead, in which case it should eliminate sx entirely.

There are other cases involving two assignments in a loop in which we would report V as current when V is endangered. I expect them to occur rarely, but I do not know that they are all semantically pathological. Copperman [1993] delimits the cases in which this kind of nonconservative error can occur. They involve assignments within a loop, affected by optimization in such a way that some variable they both can affect is current if the loop is traversed once, but endangered if the loop is traversed more than once. The compiler must have determined that all variables potentially affected by both assignments are dead in order to perform the optimization.

This source of nonconservative error could perhaps be eliminated by recording in the ds-list-pairs the last two duplicate definitions or stores that occur within a loop, rather than the last one, but that would introduce conservative error in the commonly occurring case of a loop-invariant code motion. To eliminate both the nonconservative error and the error in the case of loop-invariant code motion, an algorithm must distinguish transparent assignments whose left-hand sides are loop invariant from those whose left-hand sides are not. Perhaps future research will uncover a method of distinguishing these and using the information felicitously. Until that time, I prefer the nonconservative error because (1) the algorithm is more elegant, (2) errors will occur less frequently, and (3) the errors that do occur occur in possibly pathological cases involving pointers.

7. COST

There are four distinct costs that might be considered relevant to evaluating this method of currency determination:

(1) the cost of either not debugging optimized code at the source level or being occasionally misled by the debugger,
(2) the engineering cost of modifying a compiler and debugger,
(3) the additional space and time the compiler takes to produce the DS-graph, and
(4) the additional space and time the debugger takes to run the currency determination algorithms.

The first should be weighed against the rest to determine whether the enterprise is worthwhile. Unfortunately, they are not comparable, hard to measure, and are incurred by different groups. For the most part, academia and industry did not consider the enterprise worthwhile until the late 1980s or early 1990s. Currently there is activity in both academia and industry. If the balance of these costs has shifted, it can be seen as a continuation of the trend of the cost of programmers rising relative to the cost of computers, coupled with improved optimization technology and widespread availability of that technology.

The cost of not debugging optimized code at the source level includes programmer frustration, time spent recompiling to enable and disable optimization, disk space to store both optimized and unoptimized object modules and executables, time spent communicating with compiler vendors incorrectly claiming the compiler has a bug, time spent debugging the wrong thing because the debugger gave misleading information, and time spent finding bugs via assembly-level debugging (minus the time it would have taken via source-level debugging, were that an option). Quantitative measures of these are not available.

The engineering cost of modifying a compiler and debugger is also not available. In general, it depends on the organization of the compiler and debugger that are to be modified and the capabilities of the engineers that do the modification. No implementation has been done to date, and any estimate I made would be subject to the same poor correlation between estimates and actual costs of software production that is endemic to the industry.

The additional space and time a compiler needs to produce the DS-graph can be estimated with greater confidence.

7.1 Cost to Produce the DS-Graph (During Compilation)

The space used by DS-graph is linear in the size of the object graph. The time to construct the DS-graph is linear in the size of the object graph as well.

7.2 Cost to Use the DS-Graph (After Compilation)

7.2.1 *Space.* Currency determination increases space usage by the size of the In, Out, and PRS sets. The space usage of the In and Out sets dominates that used by the PRS sets, because there is an In and Out set per block. Let n be the number of nodes in the DS-graph and m be the number of assignments to V in the program.

For Algorithms PRS, the In and Out sets can take $O(nm^2)$ space. These sets can be constructed as needed, so the total space for In and Out sets is $O(qnm^2)$, where q is the number of variables about which the user queries.

For Algorithm PRS^T, the In and Out sets can take $O(nm(m!^2))$ space, because the number of elements in In_B^V is $O(m!^2)$, giving a worst-case bound on the space for In and Out sets of $O(qnm(m!^2))$. The bound on the number of elements in In_B^V is derived by taking the number of permutations of definitions and multiplying it by the number of permutations of stores. In the expected case, the number of definitions and stores that reach B is limited, and the order in which they reach along different paths will include few of the possible permutations. The squaring of $m!$ is from allowing definitions and stores to be ordered independently. But they are not independent—a store can only move via optimization. If many definitions or stores reach along some path, then there are many assignments through pointers, and the optimizer probably can not move many of them. I believe in practice that the number of elements in any In set will be small, and would be surprised if it exceeds m.

7.2.2 *Time.* Algorithms PRS-BP and PRS-BP^T and the initialization phases of Algorithms PRS and PRS^T are inexpensive relative to the iteration phase of Algorithms PRS and PRS^T, and will not be discussed further.

The worst-case asymptotic cost of Algorithm PRS is poor, though polynomial. The algorithm is presented as being run for all variables. However, it is reasonable for the debugger to run it on a single variable. The cost described here is for a single variable.

The worst-case asymptotic cost is $O(n^3m^2)$. In practice two factors of n and a factor of m can be replaced with constant factors, for an $O(nm)$ running time.

The worst-case asymptotic cost of Algorithm PRS^T is $O(n^3m(m!^2))$ (again, because of the size of the In sets), but its expected running time is also $O(nm)$.

A detailed analysis of these asymptotic costs is given in Copperman [1993].

7.3 Parameters

The parameters that affect the cost are:

—n, the number of basic blocks in a routine,

—m, the number of assignments to a variable within a routine, including assignments through pointers that might point to that variable,

—the number of paths to a block, and

—the number of predecessors and successors per block.

These depend considerably on program characteristics and coding style. In particular, because each subroutine is a flow graph component, the cost increases with the size of subroutines. The cost also increases with the use of pointers.

8. OPEN PROBLEM: WHEN A BREAKPOINT HAS MOVED

Under the breakpoint model given in Section 2, there is no guarantee that a semantic breakpoint is reached in optimized code if and only if it would be reached in unoptimized code. The following situations can arise with a semantic breakpoint for a statement S:

(1) The code for S has not been moved. The semantic breakpoint is the same as the syntactic breakpoint, and no additional work is required for currency determination.

(2) The code for S has been moved. In a particular execution, the semantic breakpoint location and the syntactic breakpoint location are reached along the same path.

(3) The code for S has been moved. In a particular execution, the syntactic breakpoint location is reached, but the semantic breakpoint location is not. This is unexpected behavior, but no additional work is required for currency determination at the semantic breakpoint, because it is never reached.

(4) The code for S has been moved. In a particular execution, the semantic breakpoint location is reached, but the syntactic breakpoint location is not. This is unexpected behavior already.

An approach taken by Berger and Wismüller [1993] is to use a more flexible mapping between source statements and breakpoints. They attempt to map a source statement to a breakpoint location in such a way that the breakpoint is reached if and only if it would be reached in unoptimized code.

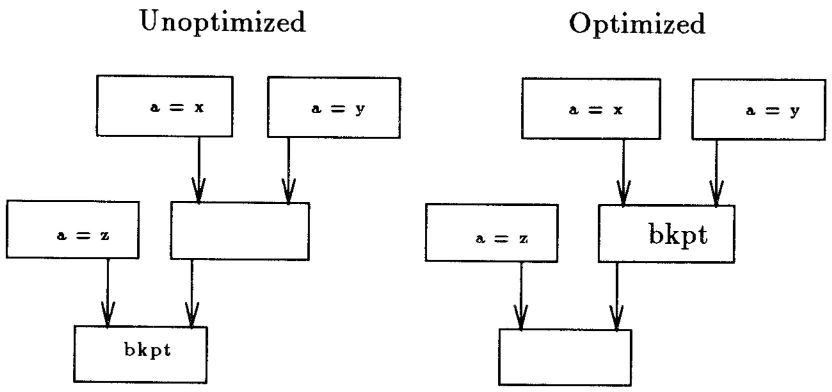


Fig. 20. Oddly enough, a is current at bkpt.

In situations 2 and 4 (above) we want to determine whether the actual value of a variable at a representative instruction R (the semantic breakpoint, where the user examines the value) can differ from its expected value at a representative instruction $R' \neq R$ (the syntactic breakpoint, where the user expects to be examining the value). Note that in general a debugger cannot distinguish situation 2 from situation 4.

Even in situation 2, there is a problem arising from the fact that the user's expectation is not well defined if the location at which a variable is examined is different from the location at which the user assumes it is being examined. Consider Figure 20. Should a debugger claim that a is current at bkpt? For bkpt to be reached in the optimized code, one of the right-hand paths must be taken. If the unoptimized code is run on the same inputs, one of the right-hand paths will be taken, so optimization does not affect the value that a will have at the semantic breakpoint for bkpt, suggesting that a is current at bkpt.

9. SUMMARY

The mapping between statements and breakpoints used for unoptimized code is problematic for optimized code. If such a mapping is used by a debugger on optimized code, the debugger is likely to mislead the debugger user. This work has described a mapping between statements and breakpoints that provides a reasonable approximation to what the naive user would expect when that mapping is used on optimized code (and provides exactly what the naive user would expect on unoptimized code). The mapping allows the debugger user to break where a statement occurs or execute a statement at a time on a program in which statements may have been reordered and instructions generated from a statement are not necessarily contiguous. The mapping enables debugger behavior that more closely approximates the behavior provided by current debuggers on unoptimized code than other proposed mappings, and thereby neither requires debugger users to be experts on optimization nor requires users to modify their debugging strategies.

Table II. The currency determination technique is applicable in the presence of any sequential optimizations, including all of the listed optimizations, that either do not modify the flow graph of the program or modify the flow graph in a constrained manner. Blocks may be added, deleted, coalesced, or copied; edges may be deleted, but control flow may not be radically changed. As an example of an optimization that does not observe the constraints; it does not apply to a portion of a program that contains interchanged loops.

Representative Optimizations		
inlining	dead store elimination	partial redundancy elimination
code hoisting	strength reductions	local instruction scheduling
constant folding	constant propagation	global instruction scheduling
cross-jumping	copy propagation	local common subexpression elimination
loop unrolling	dead code elimination	global common subexpression elimination
	other code motion	induction-variable elimination

Using any such mapping, optimization can cause a debugger to provide an unexpected and potentially misleading value when asked to display an endangered variable. A debugger must be able to determine the currency of a variable if it is to provide truthful or expected behavior on optimized code. Other researchers have given solutions to special cases of the currency determination problem.

This article describes a general solution to the problem for a large class of sequential optimizations, including optimizations that modify the shape of the flow graph. These results hold in the presence of both local and global optimizations, including those listed in Table II, and require no information about which optimizations have been performed. Additionally, this article describes the nature of the information the debugger can provide to the debugger user when the user asks for the value of an endangered variable. An abbreviated architectural design to guide the implementation of the presented method of currency determination appears in Appendix A, and a complete architectural design appears in Copperman [1993].

For most optimizations, the results described in this article are precise (i.e., a variable claimed to be current is current; a variable claimed to be endangered is endangered, etc.). In some circumstances involving assignments through pointers, there is a trade-off between rare nonconservative results and more common conservative results, which is discussed at length in Section 6.7. There are two other circumstances in which the results are conservative:

—when a variable is current along all feasible paths but noncurrent along some infeasible path, in which case it will be claimed to be endangered,

—when a variable is endangered along some path due to an assignment through an alias, but there is no execution in which that path is taken and in which that variable is affected by that assignment.

9.1 Future Work

Currency determination at semantic breakpoints remains an open question (this topic is discussed in Section 8).

Once a debugger user has found a suspicious variable (one that due to program logic, not optimization, contains an unexpected value), the next question is "How did it get that value?" The sets of reaching definitions used for currency determination can be used in a straightforward manner to answer this question ("`x was set at one of lines 323 or 351`"). One direction for future research is how to efficiently be even more helpful, how to give responses such as "`x was set at line 566 to foo(y,z). At that point, z had the value 3.141 (set at line 370) and y had the value of 17; y was set at line 506 to y+bar(w).`" This was called *flowback analysis* by Balzer [1969] and has been investigated by others ([Miller and Choi 1991; Korel 1988]); reaching sets may be adaptable to this purpose.

Another research direction is dynamic currency determination, which is how a debugger can collect the minimal execution history information needed to determine whether an endangered variable is current or noncurrent when execution is suspended at a breakpoint. Useful in conjunction with this or as an alternative is recovery, which is to have the debugger compute and display the value that a variable would have had if optimization had not endangered the variable. Finally, an exciting possibility is extending the breakpoint model and currency determination techniques to parallel code, which is rife with noncurrent variables.

APPENDIX

An appendix to this article is available in electronic form (Postscript^{TM}). Any of the following methods may be used to obtain it; or see the inside back cover of a current issue for up-to-date instructions.

—By anonymous ftp from acm.org, file [pubs.journals.toplas.append]p1270.ps.

—Send electronic mail to mailserve@acm.org containing the line
send [anonymous.pubs.journals.toplas.append]p.1270.ps.

—By *Gopher* from acm.org.

—By anonymous ftp from ftp.cs.princeton.edu, file pub/toplas/append/p1270.ps.

—Hardcopy from *Article Express*, for a fee: phone 800-238-3458, fax 201-216-8526, or write P.O. Box 1801, Hoboken NJ 07030, and request TOPLAS-APPENDIX-1270.

ACKNOWLEDGMENTS

I am indebted to Charlie McDowell for advice and guidance through all phases of this research. I thank also the editors and anonymous referees whose comments helped to substantially improve the clarity and reduce the length of this article.

REFERENCES

ADL-TABATABAI, A. AND GROSS, T. 1993a. Detection and recovery of endangered variables caused by instruction scheduling. In *Proceedings of the PLDI'93 ACM SIGPLAN/93 Conference on Programming Language Design and Implementation*. ACM, New York

ADL-TABATABAI, A. AND GROSS, T 1993b. Evicted variables and the interaction of global register allocation and symbolic debugging. In *Proceedings of the POPL'93*, The *20th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers Principles Techniques and Tools*. Addison-Wesley, Reading, Mass.

BALZER, R M. 1969. Exdams—Extendable debugging and monitoring system. In *Proceedings of AFIPS Spring Joint Computer Conference 34*. AFIPS, Washington, D.C., 125–134.

BERGER, L. AND WISMÜLLER, R. 1993. Source-level debugging of optimized programs using data flow analysis. Dept. of Computer Science, Munich Inst. of Technology, Munich, Germany

BROOKS, G., HANSEN, G. J., AND SIMMONS, S. 1992. A new approach to debugging optimized code. *SIGPLAN Not. 27*, 7 (June), 1–11.

COHN, R. 1991. Source level debugging of automatically parallelized code *SIGPLAN Not. 26*, 12 (Dec.), 132–143.

COOL, E. L. 1992 Debugging vliw code after instruction scheduling. Master's thesis, Tech Rep. CS/E 92-TH-009, Oregon Graduate Inst., Beaverton, Oreg.

COPPERMAN, M. 1993. Debugging optimized code without being misled. Ph.D. thesis, Tech. Rep. UCSC-CRL-93-21, Computer and Information Sciences, Univ. of California at Santa Cruz, Santa Cruz, Calif.

COPPERMAN, M. 1992. Debugging optimized code: currency determination with dataflow. In *Proceedings of the Supercomputer Debugging Workshop*. Los Alamos National Laboratory.

COPPERMAN, M AND MCDOWELL, C. E. 1993. A further note on hennessy's symbolic debugging of optized code. *ACM Trans. Program. Lang. Syst. 15*, 2 (Apr.), 357–365.

COPPERMAN, M. AND MCDOWELL, C. E. 1991. Debugging optimized code without surprises. In *Proceedings of Supercomputer Debugging Workshop*. Los Alamos National Laboratory.

COUTANT, D., MELOY, S., AND RUSCETTA, M. 1988 Doc: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* ACM, New York, 125–134.

FEILER, P. H. AND MEDINA-MORA, R. 1980 An incremental programming environment. Tech. Rep., Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pa.

GUPTA, R. 1990. Debugging code reorganized by a trace scheduling compiler. *Struct. Program 11*, 3 (July), 1–10.

HENNESSY, J. 1982. Symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst. 4*, 3, 323–344.

KOREL, B. 1988. Pelas program error-locating assistant system. *IEEE Trans. Softw. Eng. 14*, 9 (Sept.), 1253–1260.

MILLER, B AND CHOI, J. 1991. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst. 13*, 4, 491–530.

PINEO, P. P. AND SOFFA, M. L. 1991. Debugging parallelized code using code liberation techniques. *SIGPLAN Not. 26*, 12 (Dec.), 108–119.

POLLOCK, L. L. AND SOFFA, M. L. 1992 Incremental global reoptimization of programs. *ACM Trans. Program. Lang. Syst. 14*, 2, 173–200.

POLLOCK, L. L. AND SOFFA, M. L. 1988 High level debugging with the aid of an incremental optimizer. In *Hawaii International Conference on System Sciences*. ACM, New York.

SHU, W. S. 1993. Adapting a debugger for optimized programs. *SIGPLAN Not. 28*, 4 (Apr.), 39–44.

SHU, W. S. 1989. A unified approach to the debugging of optimized programs. Ph.D. thesis, Dept. of Computer Science, Univ. of Nottingham, U.K.

SRIVASTAVA, A. 1986. Recovery of noncurrent variables in source-level debugging of optimized code. In *Foundations of Software Technology and Theoretical Computer Science 6th Conference Proceedings*. Lecture Notes in Computer Science, vol. 241. Springer-Verlag, New York.

STREEPY, L. 1991. Cxdb a new view on optimization. In *Proceedings of the Supercomputer Debugging Workshop*. Los Alamos National Laboratory.

WALL, D., SRIVASTAVA, A., AND TEMPLIN, R. 1985. A note on Hennessy's symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan.), 176–181.

WARREN, H. S., JR. AND SCHLAEPPI, H. P. 1978. Design of the fds interactive debugging system. IBM Res. Rep. RC7214, IBM, Yorktown Heights, N.Y.

ZELLWEGER, P. 1984. Interactive source-level debugging of optimized programs. Res. Rep. CSL-84-5. Xerox Palo Alto Research Center, Palo Alto, Calif.

ZELLWEGER, P. 1983. An interactive high-level debugger for control-flow optimized programs. *SIGPLAN Not. 18*, 8 (Aug.), 159–172.

ZURAWSKI, L. W. AND JOHNSON, R. E. 1990. Debugging optimized code with expected behavior. Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Ill.

Received February 1992; revised June 1993 and July 1993; accepted July 1993