Debugging Optimized Code Without Being Misled

Max Copperman

92-01
May 8, 1992

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Optimizing compilers produce code that impedes source-level debugging. Examples are given in which optimization changes the behavior of a program even when the optimizer is correct, showing that in some circumstances it is not possible to completely debug an unoptimized version of a program. Source-level debuggers designed for unoptimized code may mislead the debugger user when invoked on optimized code. One situation that can mislead the user is a mismatch between where the user expects a breakpoint to be located and the breakpoint's actual location. This mismatch may occur due to statement reordering and discontiguous code generated from a statement. This paper 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 closely enough that the user need not make severe changes in debugging strategies. Another situation that can mislead the user is when optimization has caused 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 paper gives and proves 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. The information a compiler must make available to the debugger for this task 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: Algorithms, Languages
Additional Keywords and Phrases: debugging, compiler optimization, reaching definitions, noncurrent variables

```
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;
...                         ...                            ...
```

Figure 1.1: Potentially Confusing Optimizations

1 Introduction

A source-level debugger should have 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, presumably the user wishes to examine the state of the program, often by querying the value of a variable V. Commonly available debuggers, upon receiving such a query, will display the value in V's storage location. Unfortunately, this value may be misleading due to optimization. For example, due to a code motion optimization, an assignment to V may have been done earlier than the source code would lead one to expect. Since one aspect of debugging is examining potential anomalies, the debugger user may expend time and effort attempting to determine why V contains the value that has been displayed when the source code suggests that V should contain some other value.

Figure 1.1 is an example of such a situation caused by constant propagation followed by dead store elimination. 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`, 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`.

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.

The source-level debugger user probes the state of a halted executable while looking at the source code from which it was compiled. Much of the user's activity consists of inference based on the source code and the state information provided by the debugger. This state information includes the location at which execution is halted and the values of variables. One implicit assumption is that the value of each variable in the halted executable corresponds one-to-one to 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. Another implicit assumption is that the location at which execution is halted corresponds to a location in the source code specified by the user. These assumptions may be violated by the presence of optimization, and the inferences that the user draws may be incorrect.

This undesirable situation may sometimes be avoided by disabling optimizations when debugging.[1] At best this is inconvenient, because it requires extra compilation steps. At worst, however, it may be impossible.

Optimization May Change the Behavior of a Program

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. Optimizations are correctness-preserving transformations which, if the compiler is correct, will not change the behavior of a correct program. However, a program that is being debugged is certifiably not a correct program, and correctness-preserving transformations are not guaranteed to preserve the behavior of an incorrect program.

It is a misconception that if optimization changes the behavior of a program, the compiler must be incorrect.[2] There are two circumstances in which correct optimization may change the behavior of a program.

- 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, but there are others. A correct optimized translation of a program containing code with loose semantics may have different behavior from a correct unoptimized translation of that program.
- Buggy programs:
  A correct optimized translation of a program containing a bug may have different behavior from a correct unoptimized translation of that program. This is a commonly overlooked case that is important because a program that is being debugged is known to have bugs.

It is common for even experienced software engineers to be surprised at the fact that a program can behave one way when optimized and a different way when unoptimized *when it has been compiled with a correct compiler*. Consider the compiler writer's perspective:

It is no sin to make a wrong program worse.

— W. M. McKeeman

Figures 1.2 and 1.3 are simple examples of programs with bugs. The following text describes how optimization can affect them.

In Figure 1.2, the bug is benign when the program is unoptimized, but has an effect when unnecessary stores are eliminated. The bug is to write past the end of an array, overwriting the character `c`. In the absence

[1] There is at least one highly optimizing compiler [Pic90] that, when compiling with optimizations disabled, still performs live/dead analysis, constant propagation, copy propagation, and global register allocation, any of which can confuse a source-level debugger.

[2] If the compiler 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 a fixed compiler, the programmer typically has to debug the optimized code enough to convince the compiler vendor that it is a compiler bug.

```
int i;
char a, b[10], c;

void overwrite_c() {
    a = getchar();
    c = a;
    for (i=0; i<=10; i++) {
        b[i] = '\0';
        }
    c = a;
    if (c == '\0') {
          program misbehaves
        }
    }
```

Figure 1.2: Optimization Changes Program Behavior: Example 1

of optimization, c is subsequently set to its previous value and the bug goes unnoticed. In the presence of optimization, the bug affects the behavior of the program: the optimizer eliminates the second assignment into c because it can determine that in a correct program, c would already contain the to-be-assigned value.

Note that this situation can occur with statements that are arbitrarily far apart in the source code, so long as the optimizer can determine that a has not been modified between the first and second assignment into c.

In Figure 1.3, the bug (writing one byte past the end of b) has an effect when the program is not optimized. It is benign ("goes away") when data fetches are optimized by aligning data structures on 4 byte boundaries.[3] If each data object is aligned to a four byte boundary, there will be two bytes of padding between the end of array b and character c, and the bug will have no effect on program behavior. If data objects are not aligned, there will be no padding between b and c; c will be overwritten.

Optimization can change the behavior of a program. It is therefore necessary, upon occasion, to either debug optimized code or never optimize the code. Were it not for the confusion optimization can introduce into debugging, debugging optimized code would generally be preferable to debugging unoptimized code and adding recompilation steps.[4] Zellweger [Zel84] introduced terms for two methods of removing or ameliorating the confusion introduced into the debugging process by optimization. The preferred method 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. This is known as providing *expected* behavior. 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

[3]Note that program misbehavior, which is the external evidence of the bug, could occur when the program is *not* optimized and go away when the program *is* optimized by changing the sense of the conditional — that is, by adding another bug. What is important is that the behavior changes depending on the presence or absence of optimization.

[4]The additional recompilation step associated with debugging unoptimized code is not a one-time cost for the program, but a cost for each debugging session.

```
int i;
char b[10], c;

void walk_on_c() {
    c = getchar();
    for (i=0; i<=10; i++) {
        b[i] = '\0';
        }
    if (c == '\0') {
          program misbehaves
        }
    }
```

Figure 1.3: Optimization Changes Program Behavior: Example 2

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. There is a range of truthful behavior, some useful and some not so useful. Appropriate truthful behavior is discussed in Section 2. Some solution to the problem addressed in this paper (exemplified by Figure 1.1) is necessary for providing expected or truthful behavior.

Approaches to the Problem

General approaches to the problem have been:

- to restrict the optimizations performed by the compiler to those that do not provoke the problem ([WS78], [ZJ90]),
- to recompile, without optimization, during an interactive debugging session, the region of code that is to be debugged ([FM80], [ZJ90]), and
- 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 ([WS78], [Hen82], [Ze83a], [Ze83b], [Zel84], [CMR88], [PS88], [Cop90], [ZJ90], [PS92]).

A larger problem is lowering the cost of debugging production quality software. Much if not most production quality software produced in this country is heavily optimized, and the first approach would result in compilers that would not get used; their use would degrade the quality of the software. The second approach 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 unacceptable because optimization may change the behavior of the program.

This work follows the third approach. Some of the 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. Section 7 defines one possible interface between a compiler and a debugger for the problem addressed in this paper. If such an interface is used, the debugger need not be specialized to a particular set of optimizations.

Overview

If the value in a variable's storage location is suitable to be displayed to the user, the variable is *current*. The remainder of this paper describes how to determine whether a variable is current at a breakpoint – the problem of *currentness determination*, first introduced by Hennessy [Hen82]. The fundamental idea behind our solution to the currentness 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.[5] The required sets of definitions of V that reach any point in a program (optimized or unoptimized) can be computed using slight modifications of standard compiler technology (Aho and Ullman [AU77]). If the sets of definitions of V that reach P differ in the optimized and unoptimized version of the program, then V is not current. The debugger can use the sets of definitions to describe, in source-level terms, why V is not current. Unfortunately, if the two sets of definitions are equal it is still possible that V is not current. This is discussed further in Section 4.3.

In order to determine a variable's currentness:

1. The compiler must generate a set of debug records relating statements to code addresses; these debug records are ordered in two flow graphs, one representing the program before optimization and the other representing the program after optimization.
2. The flow graphs are used to compute reaching definitions, which are in turn used to create reaching sets (sets of definitions that reach a breakpoint location).
3. The reaching sets are compared to compute the currentness of variables.

The focus of the paper is how reaching sets are used to compute the currentness of a variable at a breakpoint.

Section 2 defines a breakpoint model that is appropriate for optimized code. The determination of appropriate breakpoint locations is discussed, as are appropriate debugger capabilities at a breakpoint, because these are more complex than their counterparts for unoptimized code.

Section 3 defines the terminology that is used throughout the rest of this work. This section also provides motivational examples.

Section 4 describes a solution to the problem of *currentness determination*. Such a solution is necessary to provide the debugger capabilities discussed in section 2. It also describes how this solution can be used to provide helpful truthful behavior when expected behavior cannot be provided.[6]

[5] One compilation of the program is sufficient to provide the information with which to compute both the definitions that ought to reach P and those that actually reach P.

[6] It is trivial to provide (useless) truthful behavior. Simply always give a warning that the code has been optimized. This is of course unacceptable.

Section 5 shows the correctness of the solution.

Section 1 briefly describes a problematic special case.

Section 7 describes the data structures that must be produced by the compiler (the debug records and flow graphs).

Section 8 summarizes and discusses the accuracy of the results.

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.[7] 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. Users not versed in optimizing technology expect these characteristics to hold when execution is suspended at a statement boundary.

The state of a suspended program is the context in which debugging takes place, called the *actual debugging context*. In contrast, the *expected debugging context* is the state that would be predicted by an examination of the source code of a program suspended at an identified point. The actual debugging context matches the expected debugging context for an unoptimized program suspended at a statement boundary.

2.1 Treatment of Program Traps

The actual debugging context may not match the expected debugging context, even for unoptimized code, if the program halts on a non-statement boundary, which can happen due to a trap (an error condition).[8] A program may trap in the middle of an update to a variable, leaving that variable in a decidedly unexpected state. The most important piece of information when a program traps is "What statement caused the trap?", that is, which statement generated the instruction that trapped. This information can be provided by tagging each instruction with a reference to the statement that generated it. This can be encoded in a table by listing the address of the first instruction of each set of contiguous instructions generated from a source statement with a reference to that source statement, thus the trap location reporting problem can be solved by a simple extension of the line table currently emitted by most compilers. The remainder of this paper considers programs that are suspended at source-level user-specified locations (breakpoints) only.

2.2 Debugger Capabilities at a Breakpoint in Optimized Code

Optimization may well make it impractical to provide the user with the expected debugging context. Because code may be reordered or eliminated and the instructions generated from a given source statement

[7] 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 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.

[8] A program can also halt on a non-statement boundary due to machine-level debugging – single stepping machine instructions or breaking at arbitrary code addresses. This work is concerned with source-level debugging, not machine-level debugging. Sections 2.3 and 2.4 define allowable breakpoint locations for source-level debugging.

may not be contiguous, when execution is suspended at statement S in optimized code, no matter what code location is chosen to represent S, some of the code from previous statements may not yet have been executed and some of the code from subsequent statements may have been executed early.

The debugger user makes inferences based upon the source code and the state of the halted program. This is problematic for debugging optimized code because the inferences are also based upon the implicit assumption that the actual debugging context is equivalent to the expected debugging context.

Of course, it is not possible to prevent a user from making invalid inferences, regardless of the presence of optimization. The best the debugger can do is provide a means of determining when optimization has broken an otherwise valid chain of inference, that is, when an inference that would be valid in the absence of optimization is invalid in its presence. To this end, the debugger acts satisfactorily upon optimized code if at a breakpoint it can report the ways in which the actual debugging context differs from the expected debugging context.

At a breakpoint, the user should be informed of salient differences between the actual debugging context and the expected debugging context. If the user asks to see the value of a variable, the debugger should offer information as to whether its value would be misleading, and why. The user should be able to ask whether a given statement has been executed out of order, and if so, whether it has been executed early or will be executed late. These capabilities allow the user the same power to probe the state of an optimized program at a breakpoint that is available currently for unoptimized programs, because they license valid inferences based on the source code and the state of the suspended program and they provide information that can be used to prevent invalid inferences.

Only those effects of optimization that affect the validity of the user's inferences need to be reported by the debugger. As noted by Coutant et al [CMR88], much of the optimization performed upon a program is irrelevant to the user. It is only optimization that affects user-visible entities, such as source code variables and statement flow-of-control, that the user needs to be informed about. Informing the user of optimization on compiler temporaries is likely to make the debugging job harder, not easier. The same is true of optimization of code generated from the right-hand-side of assignments – the store of the result affects the state of the program as seen from the source-level view, but how that result is computed does not affect the source-level view of program state. Similarly, optimization of an expression whose result determines the outcome of a conditional branch should be invisible to the user if the branch itself is unaffected.[9] Many statements that start earlier in optimized code than in unoptimized code do so due to code motion of parts of the statements (such as address computations) that are irrelevant to the user's inquiry.[10] Though the optimization of these statements does cause the actual debugging context to differ from the expected debugging context, it does not invalidate user inferences, therefore it is not necessary for the debugger to

[9] There are circumstances in which it is important for the debugger to reveal the effects of optimization at this level of detail, such as allowing the user to track down a code-generation bug. In such circumstances, it is appropriate to shift to machine-level debugging.

[10] Note that this code motion is not irrelevant to trap location reporting. If an address computation is moved up out of a loop, and the computation traps, the user should be informed that the trap occurred in the statement that the address computation originated in.

report that these statements have begun early. Statements that begin early due to source-level-invisible optimization but that otherwise exhibit no source-level-visible effects from optimization are not considered to be executed out of order.

2.3 Breakpoint Locations (Representative Instructions)

Commonly, when setting a breakpoint on a statement, the debugger user wants to break exactly once each time the statement is executed at some location that corresponds to the statement boundary. This is problematic for optimized code, but not providing or closely approximating this capability puts a heavy burden on the user not well-versed in optimizer technology. The capability is necessary to support two common debugging strategies: running until a selected statement is reached, and stepping through the program statement by statement.

In Section 2.1's treatment of program traps, every instruction generated from a statement is associated with that statement. This is possible and appropriate because the program may trap at an arbitrary location that is mapped back to the source code. A breakpoint is specified in source terms and must be mapped onto the machine code. It is inappropriate to associate every instruction generated from a statement with that statement for the purposes of setting breakpoints, because if the instructions are not contiguous, many breakpoints may be reached for a single statement. In contrast, Streepy [Str91] describes 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, when a statement is selected as the level of granularity, a breakpoint is set at the beginning of each sequence of contiguous instructions generated from the statement. Under the mapping described in this section, the instruction generated from a statement S that best corresponds to the statement boundary is selected to represent S, and is called the *representative instruction* for S. The address of this instruction is a breakpoint location for S.[11] Where no confusion will result, the representative instruction itself may be referred to as the breakpoint location. The mapping described herein is not in conflict with that described by Streepy [Str91]; each enables debugger capabilities missing from the other. This paper does not concern itself further with breakpoints for language entities other than statements, except to state that the results hold in the presence of such breakpoints.

The choice of a machine instruction as the breakpoint location for a statement should be based on why the user wants to break at that statement. It may be that the user sets a breakpoint at some statement within a loop because it looks like a convenient place to see how the program state is changing on subsequent iterations of the loop. There may be nothing about the chosen statement relevant to the user's purpose except its location within the loop. If that statement were moved out of the loop by optimization, it would be appropriate to set the breakpoint where it used to be, so the breakpoint would be reached each time

[11] In the most common case, a single instruction will serve as the breakpoint location for a statement. Statements with multiple side effect on user variables will require multiple breakpoint locations, one for each side effect. Optimizations that cause code duplication may require breakpoint location duplication as well – procedure integration (inlining), partial redundancy elimination, and loop unwinding are examples. Even in unoptimized code some statements may require more than a single instruction to represent their breakpoint locations. Loop constructs are an example. The appropriate location to break the first time (before the loop is entered) may be at a different instruction than the appropriate location to break subsequently (each time through the loop).

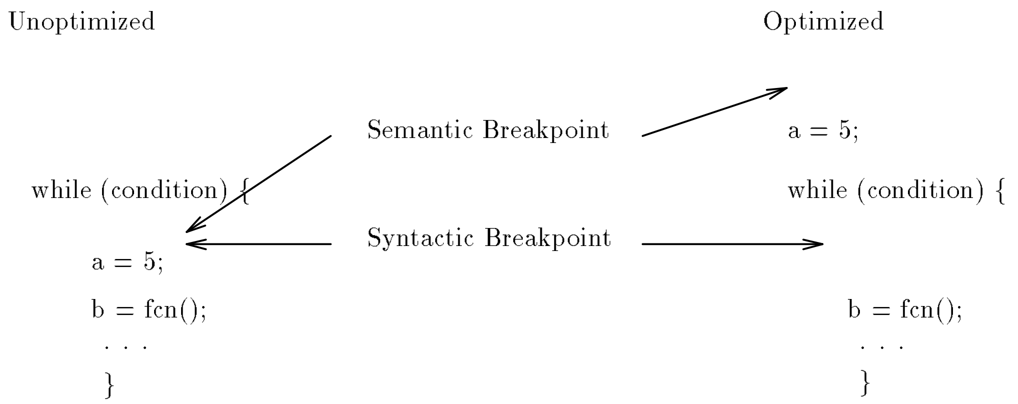


Figure 2.1: Semantic and Syntactic Breakpoint Locations

through the loop. On the other hand, the user may set a breakpoint at some statement to check the values of variables used in an expression in that statement. In that case, if the statement were moved out of the loop by optimization, it would be appropriate to set the breakpoint where it ended up, so the values the debugger displays are the actual values used in the expression.

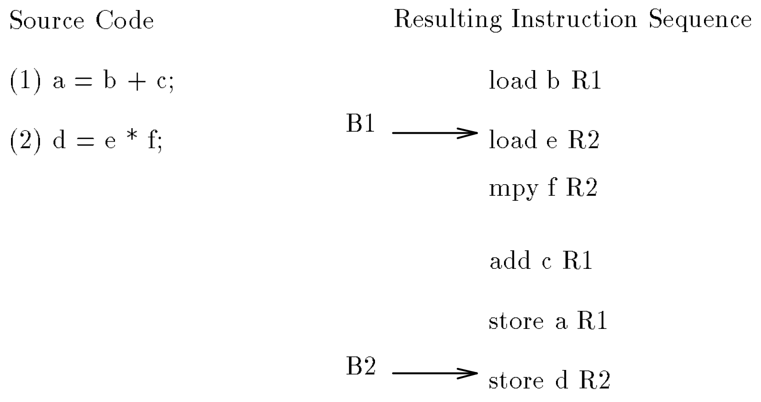
Zellweger [Zel84] introduced the terms *syntactic* and *semantic* breakpoints. If no code motion or elimination has occurred, these are identical. In the presence of code motion or elimination, the order in which syntactic breakpoints are reached reflects the syntactic order of source statements; the syntactic breakpoint for statement n is prior to or at the same location as the syntactic breakpoint for statement $n + 1$. It will be at the same location if the code for n is moved or eliminated. If the code generated from statement n is moved out of a loop, a syntactic breakpoint for n remains inside the loop.[12]

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 a statement is contiguous, the semantic breakpoint location is the location at which the code for the statement has ended up. If the code generated from statement S is discontiguous, the semantic breakpoint location is the location at which the instruction chosen to represent S has ended up.

Figure 2.1 provides an example of the syntactic and semantic breakpoints for a loop from which optimization has moved an invariant statement.

The choice of a breakpoint location for a statement S affects the correspondence between the actual debugging context and the expected debugging context considerably. Zellweger [Zel84] has a discussion of possible semantic breakpoint locations for statements whose generated code is discontiguous. The view taken in this work is that the best breakpoint location for a programming language construct is the location that corresponds most closely to the source level view of the program. The breakpoint location for a statement should be the address of the instruction that most closely reflects the effect of the statement on user-visible entities (program variables and control flow). For each construct in a programming language, the breakpoint

[12]There are circumstances under which a syntactic breapoint for a statement may be undefined. Section 7.4 describes how syntactic and semantic breakpoint locations are determined under the breakpoint model summarized in Section 2.4.



```
Source Code                    Resulting Instruction Sequence

(1) a = b + c;                      load b R1

(2) d = e * f;        B1 ------->   load e R2
                                    mpy f R2

                                    add c R1

                                    store a R1

                      B2 ------->   store d R2
```

Figure 2.2: Breakpoint Location Choices for Statement (2)

location (equivalently, the representative instruction) should be chosen appropriately.

For statements involving program-variable updates, the instruction that stores into the variable is the right choice.[13] This is illustrated by figure 2.2, which gives a fragment of source code and an optimized sequence of instructions that could result. One might want to break at statement (2) and examine `a`. If the first instruction generated from a statement is the representative instruction for that statement, a breakpoint at statement (2) would suspend execution at B1, resulting in examining `a` when it has not yet had `b + c` stored into it. If, instead, the store instruction (B2) is the representative instruction for an assignment, the breakpoint will be reached at B2 and the store into `a` will have occurred.[14]

For control-flow statements (branching or looping constructs), the instruction that accomplishes the control transfer (typically a conditional branch) is the appropriate choice; it provides a natural sequence point for program dependences. Consider the code fragment in figure 2.3. The computation of `(b + c * d)` can be computed before the assignment into `a`, however, the jump to the **then** or **else** case must follow the assignment if correctness is to be maintained.

2.4 A Summary of the Proposed Breakpoint Model

A debugger may have the capability of suspending the execution of a program at an arbitrary instruction. The results described in the remainder of this paper do not hold at arbitrary instructions. The points at which the results hold are termed *valid* breakpoints and constitute the breakpoint model used in the remainder of the paper. The set of valid breakpoints is the set of representative instructions as described above: for a variable modification that appears in the source code, the store into the variable is the associated breakpoint. An assignment that has side effects will have more than one associated breakpoint. For branching and looping constructs, the branch instruction is the associated breakpoint. The C statement

[13] A "store" in this context need 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 accomplishes the action of the source statement.

[14] If the optimizer has reversed the order of the stores into `a` and `d`, then there is no way to choose a representative instruction for statement (2) that gives expected results; either `a` or `d` will have an unexpected value.

```
Unoptimized              Optimized

                          R1 = b + c * d
 a = x;                   a = x;
 if (b + c * d)           if (R1)
     e = a;                   e = a;
 else                     else
    e = -a;                  e = -a;
```

Figure 2.3: The Branch is a Sequence Point for Dependences

```
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. Choosing the store as the breakpoint location for variable modifications is crucial to the correctness of the work presented in the remainder of the paper. Additional breakpoints, such as those described by Streepy [Str91], could easily be incorporated into this model.

The remainder of this paper assumes only syntactic breakpoints are available, because space constraints do not permit a complete discussion of the additional complexity needed to handle semantic breakpoints. Section 1 briefly discusses the problems raised by semantic breakpoints. However, the proposed breakpoint model supports both syntactic and semantic breakpoints. This does not increase the number of breakpoint locations, but it affects the mapping between source-level specifications of breakpoints and breakpoint locations. A source-level specification of a breakpoint is a specification of its type (syntactic or semantic) and a reference to a statement or side effect within a statement. This work does not specify a user interface, so it does not describe the form of such a reference.[15]

[15] An implementation could accept a statement reference (such as a line number) and set breakpoints at every valid breakpoint contained therein. The user would not need to specify the type of breakpoint nor the side effect within a statement. However, for some statements the debugger would gain control more than once during the execution of the statement, and the location at which the debugger gains control may not be the location the user expects. As always, the debugger should provide enough information that the user is not misled. The advantage of this scenario is that user that is naive about optimization can still use the debugger effectively. The debugger could even gently educate the naive user about the different types of breakpoints.

3 Currentness

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 examining 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. Abstractly, this would-be-predicted value is the value that would be given to the variable if the program were running on a machine whose instruction set is the source language.

In unoptimized code, at each breakpoint the expected value of every variable is identical to its actual value. In optimized code, as we have seen, the actual value of a variable at some point may differ from its expected value at that point. Hennessy [Hen82] introduced the terms *current*, *noncurrent*, and *endangered* to describe the relationship between a variable's actual value and its expected value at a valid breakpoint. This relationship is described on the basis of a static analysis, one that inherently cannot use information about how the breakpoint was reached.

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. Examples of current variables are given in Figures 3.1 and 3.2. All examples use program flow graphs. Nodes in the flow graphs represent basic blocks and edges represent basic block connectivity. For clarity of exposition, the example graphs are minimal (for example, there is at most one instruction within a basic block), and thus they describe programs that do nothing interesting. The language of the examples includes assignment (`a = x` 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.

The examples are better understood as flow graph pieces that contain all the relevant information about a variable at a breakpoint. Thus an example flow graph is representative of the family of flow graphs that contain the example graph with arbitrary other edges, nodes, and instructions, so long as these additional elements do not change which definitions of shown variables reach shown points within the example graph.

Figure 3.1 shows the simplest case of a variable that is current at a breakpoint. There is a single assignment into `a` prior to the breakpoint, and this assignment is unaffected by optimization. There is only one way to reach **`bkpt`** in both versions of the program, and in both versions, along the only path to **`bkpt`**, `a` receives its value from the same assignment.

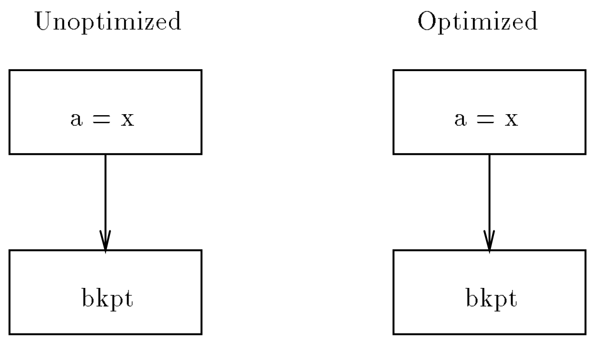


Figure 3.1: Variable `a` is current at `bkpt`: the simplest example

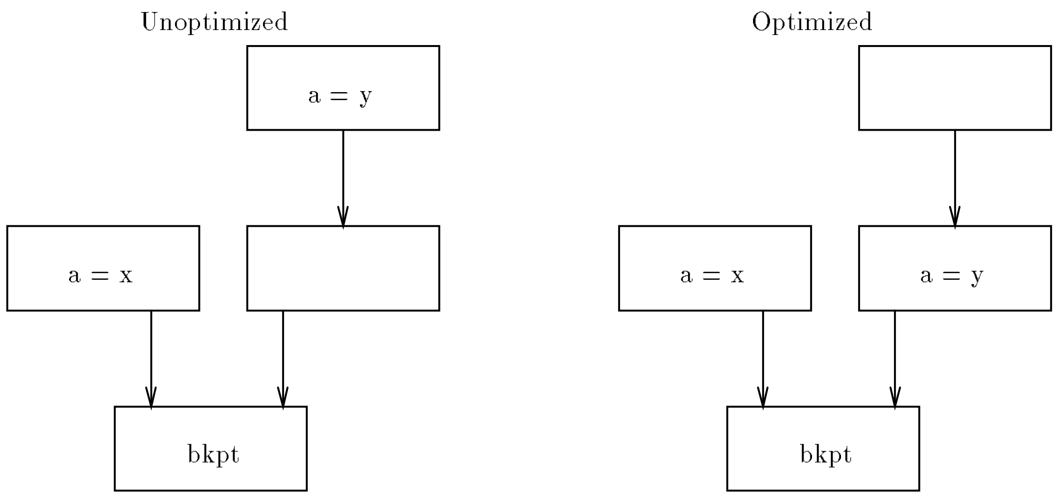


Figure 3.2: Variable `a` is current at `bkpt` in the presence of relevant optimization

A variable may be current at a breakpoint even if optimization has affected assignments into the variable. Figure 3.2 shows a case in which an assignment into `a` has been moved. Variable `a` is still 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 3.3 is a simple example of a noncurrent variable, and could be a result of dead store elimination. 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`.

Code motion can also make a variable noncurrent. In Figure 3.4, the assignment into `a` reaches `bkpt` in the unoptimized code but does not reach `bkpt` in the optimized code, thus `a` is noncurrent at `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.

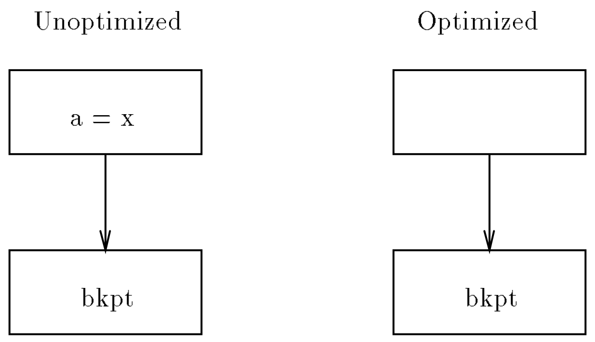


Figure 3.3: Variable `a` is noncurrent at `bkpt`: the simplest example

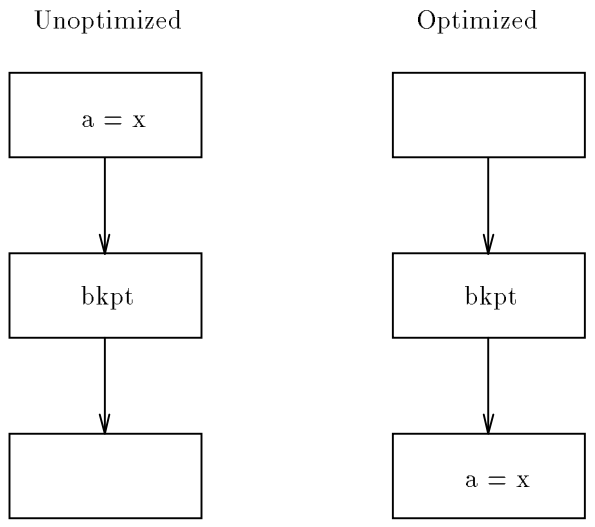


Figure 3.4: Variable `a` is noncurrent at `bkpt` due to code motion

In Figure 3.5, along the left-hand path the assignment into `a` that reaches `bkpt` in the unoptimized code corresponds to the assignment into `a` that reaches `bkpt` in the optimized code, but along the right-hand path this is not the case. `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 3.5, `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 cannot determine which, because knowledge of the path taken is absent. A debugger that does not save 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, therefore we do not assume such information is available to the debugger.[16]

In order to talk about V's currentness along a particular path, a path must be defined in such a way that

[16]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 is an open question. I term this *dynamic currentness determination.*

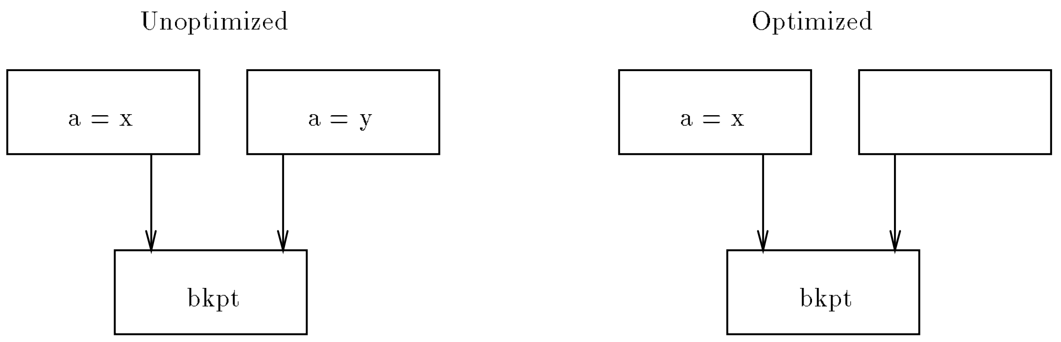


Figure 3.5: Variable `a` is endangered at `bkpt`

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* p is a pair $< p_u, p_o >$ where p_u is the sequence of basic blocks visited in an execution of an unoptimized version of a program and p_o is the sequence of logical blocks visited in an execution of an optimized version of the same code on the same inputs.

The correspondence between basic blocks in p_u and logical blocks in p_o is as follows:

1. A block b_u in p_u may have no corresponding block in p_o if optimization has caused all of the code in b_u to be moved or eliminated.
2. Those basic blocks introduced by optimization (such as loop pre-headers) have a single successor, and such a block together with its successor form a single logical block.
3. There may be one block b_o in p_o corresponding to a sequence of blocks in p_u, on condition that if the first block in the sequence in p_u is entered, execution will always proceed through the entire sequence. In this circumstance, the single block b_o is treated as a sequence of logical blocks corresponding to the sequence in p_u.[17]
4. Multiple blocks $b_1, b_2, ..., b_n$ in p_o (not necessarily contiguous) may correspond to multiple instances of a single block b in p_u, on condition that one of the b_i is in p_o iff b is at the same point in the sequence p_u. This is the correspondence needed for, say, loop unrolling or inlining (procecedure integration).
5. A block b_u in p_u has one corresponding block b_o in p_o otherwise.

These correspondences may be combined, so for example, blocks in an unrolled loop may be coalesced.

Many classical sequential optimizations modify the flow graph only in ways that maintain these correspondences. However, there is another correspondence that is desirable:

- Multiple blocks in p_o corresponding to multiple blocks in p_u (other than such correspondences derivable from the definition of a path) ; needed for, say a compiler that recognizes bubblesort and replaces it with quicksort.

[17]A method of embedding the pre-optimization block structure in the post-optimization flow graph is described in Section 4.3. This method allows the logical blocks within b_o to be distinguished.

The results given in this paper may not hold in the presence of optimizations which require the latter correspondence, While the bubblesort/quicksort example may seem far-fetched, on the one hand there are compilers that recognize statement sequences in benchmarks (even across separate compilation units!) and replace them with fast code, and on the other, debugging a production version of a program written in a very-high-level language against a working prototype would require such correspondences. We have not investigated whether parallelizing optimizations require such correspondences.

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

Definition 2: A *path p to a block B* is a subpath of a path p' where p_u is a prefix of p'_u ending in B and p_o is a prefix of p'_o ending in the logical block corresponding to that occurrence of B.

Definition 3: A *path p from block A to block B* is a subpath of a path p' where p_u is a subsequence of p'_u starting at A and ending at B and p_o is the subsequence of p'_o starting at the logical block corresponding to that occurrence of A and ending at the logical block corresponding to that occurrence of B.

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

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 generated code and optimized generated code. These distinctions are needed in this work because it compares 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, an assignment corresponds to a store, but unlike definitions, the order of occurrence of assignments in the source code may differ from the order of occurrence of stores in the machine 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 representative instruction generated from any member 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.[18]

We can now formally define some of the terms described previously.

Definition 6: *A variable* V *is current at a breakpoint* B *along path* 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.

Definition 7: V *is noncurrent at* B *along* p iff the store into V that reaches B along p_o was not generated from the definition of V that reaches B along p_u.

Definition 8: V *is current at* B iff V is current at B along each path to B.

Definition 9: V *is noncurrent at* B iff V is noncurrent at B along each path to B.

Definition 10: V *is endangered at* B if it is noncurrent at B along at least one path to B.

3.1 Assignments Through Aliases

Definitions 6 through 10 assume a single definition or store reaches a breakpoint along any path. Consider an assignment $*P$ through a pointer (or through an array element where the index is a variable). 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. This is treated more formally in [Cop90] pp. 110-112. For any language that allows such aliasing, the assumption of a single definition reaching along a given path does not hold.

Our results hold for languages that allow aliasing with one restriction on the compiler. This section describes the restriction and gives new definitions that take aliasing into account. However, for clarity of exposition, in the remainder of the paper the simpler definitions are used.

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.

[18] 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).

A problem can arise if the last store into V that reaches B along a path p is generated from a definition of V other than the last that reaches B along p, that is, if the compiler has changed the order of assignments along p. If V is live at B, changing the order of assignments into V that reach B along p changes the semantics of the program, so the problem cannot arise. However, if V is dead (but presumably some other variable that also could be assigned into by the reaching store is live), the compiler is free to change the order of such assignments.

The debugger could be burdened with determining that the order of assignments has not been changed, but it is probably preferable to restrict the compiler so that it does not change the order of such assignments. This is not a severe restriction on the compiler, because the conditions under which it is both correct and advantageous to make such changes are unlikely to occur often, and it is expensive to determine that these conditions have occurred. Under this restriction on the compiler, Definitions 6, 7 and 10 must be modified and one definition must be added as follows to preserve the correctness of our work in the presence of multiple assignments on a path:

Redefinition 1 (6): *V is current at B along path p* iff every store into V that reaches B along p was generated from a definition of V that reaches B along p.

Redefinition 2 (7): *V is noncurrent at B along p* iff no store into V that reaches B along p was generated from a definition of V that reaches B along p.

According to these definitions, V may at the same time be neither current nor noncurrent along a path. This happens when an assignment through an alias is eliminated.

Definition 11: *V is endangered at B along p* if it is neither current nor noncurrent at B along p.

Redefinition 3 (10): *V is endangered at B* if it is noncurrent or endangered at B along at least one path to B.

We turn now to the problem of how to determine whether a variable is current.

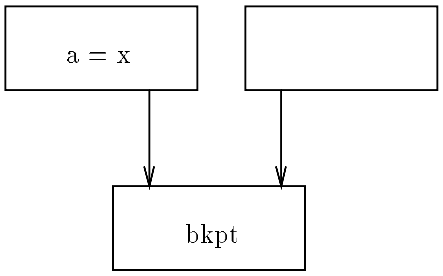


Figure 4.1: One Definition Reaches But Not On All Paths

4 Currentness Determination

This section describes how to determine which state of currentness a variable is in at a breakpoint – the problem of *currentness determination*.

Two sets of reaching definitions are needed to compute a variable V's currentness at a breakpoint B:

- the set of stores into V that reach B, that is, the modifications to V that actually reach the point at which execution is suspended, and
- the set of definitions of V that reach B, that is, the definitions of V that the user expects to have reached the point at which execution is suspended.

These sets are called *reaching sets*.

Section 7 describes the information that is needed to compute these two sets of reaching definitions. A number of variations on how to compute these sets of definitions exist, trading storage space and one-time computation costs for speed at the point of the (interactive) query. The two most straightforward are that they are pre-computed by the compiler or that they are computed by the debugger at the point of the query about V. Regardless of what tool computes them, we assume in this section that they are available.

4.1 Case Analysis

The problem is straightforward when at least one of the reaching sets is a singleton set, so the analysis of the problem is based on the cardinalities of the reaching sets. We ensure that the reaching sets be nonempty (this ensures that some definition (store) for each variable reaches a breakpoint along every path) by defining the beginning of the program or subroutine, that is, the start node of a connected component of a flow graph, to be a null definition and a null store of every variable. This also ensures that if only one definition (store) for a variable reaches a breakpoint, it reaches along all paths to the breakpoint, ruling out the situation shown in Figure 4.1.

When exactly one store into V and one definition of V reach a breakpoint B, V is current if the store was generated from the definition and noncurrent otherwise. If a single representation is used for stores and definitions (as described in Section 7) it is sufficient to compare the reaching sets.

When one of the reaching sets is a singleton set and the other is larger, comparing the reaching sets is still sufficient. Suppose one definition and many stores reach the breakpoint. At most one of the stores was

	One definition, d, reaches B	Many definitions reach B
One store, s, reaches B	Was s generated from d? Yes: current No: noncurrent	Was s generated from one of the definitions that reach? Yes: endangered No: noncurrent
Many stores reach B	Was one of the stores generated from d? Yes: endangered No: noncurrent	

Table 4.1: The Simple Cases

generated from the definition, so V is endangered.[19] If none of the stores were generated from the definition, V is noncurrent. The case in which a single store and many definitions reach the breakpoint is analogous. These three cases are summarized in Table 4.1.

In the fourth case, in which many definitions of and many stores into a variable reach a breakpoint, comparison of the reaching sets alone is not sufficient to determine a variable's currentness. The additional work that is required to make the determination is described in Sections 4.3 and 4.4. Table 4.2 summarizes this additional work. Before analyzing this more complex case, the next section briefly mentions how the debugger can decribe the effects of optimization when a variable is endangered at a breakpoint.

4.2 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 a variable a when the optimization shown in Figure 4.2 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 at one of lines 327 or 342."

The description of the effects of optimization will vary in specificity as the effects of optimization vary in complexity. The information needed to produce such messages can be made available via the reaching sets. The representation described in Section 7 provides the necessary information. Any representation of a reaching set element that provides both a source reference (such as file name and line number) and the code address of the representative instruction will do.

4.3 Multiple Stores and Multiple Definitions

Consider the case in which there are multiple definitions of V and stores into V that reach a breakpoint. If there are any stores that reach that are not generated from definitions that reach, or any definitions that

[19] Definition 5 defines multiple machine stores generated from a single definition as a single store in our terminology.

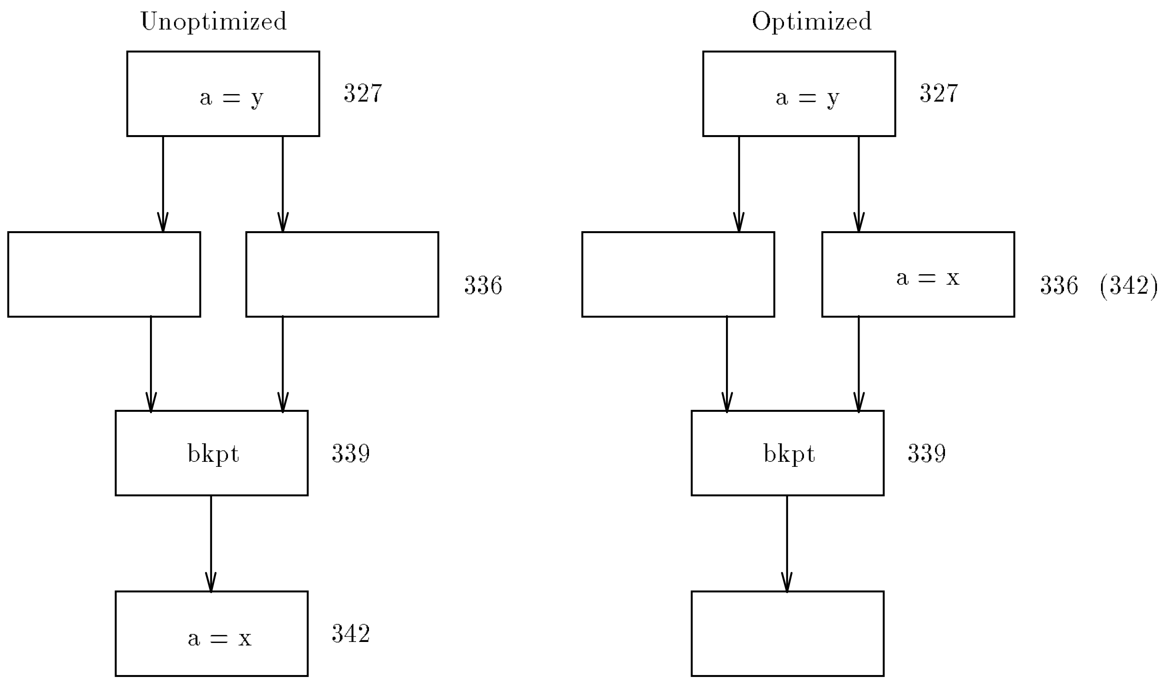


Figure 4.2: The display of a could be accompanied by this message: "Breakpoint 1 has been reached at line 339. a should have been set at line 327. However, optimization has moved the assignment to a at line 342 to near line 336. a was actually set at one of lines 327 or 342."

reach that did not generate stores that reach, V is endangered (and possibly noncurrent) at the breakpoint. Suppose the definitions of V and stores into V that reach match perfectly: every store that reaches is generated from a definition that reaches and every definition that reaches generated a store that reaches. If V were always current in this situation, comparing the reaching sets would be a complete solution to the currentness determination problem. Unfortunately, V may sometimes be endangered (possibly even noncurrent) under these circumstances. Figure 4.3 gives examples of code motion after which the reaching stores and definitions match perfectly. In one case, a is current, in another a is endangered, and in a third a is noncurrent.[20] Clearly, comparing the reaching sets is not sufficient to determine a's currentness.

It is unacceptable to be overly conservative and claim that a variable V is endangered in such a case because a debugger must provide good behavior on unoptimized code as well as on optimized code. In unoptimized code, the stores that reach are always exactly those generated from the definitions that reach. A debugger using such an algorithm on unoptimized code would claim that any variable that has definitions on more than one path to B is endangered, when in fact no variables are endangered.

Section 4.4 describes how to determine V's currentness precisely when multiple reaching stores and definitions match perfectly. The method involves examination of program flow graphs and is potentially

[20] We assume that code motion may move code up or down but not sideways, that is, the compiler will move code only to an ancestor or descendant block. This restriction is the reason for the complexity of the noncurrent example.

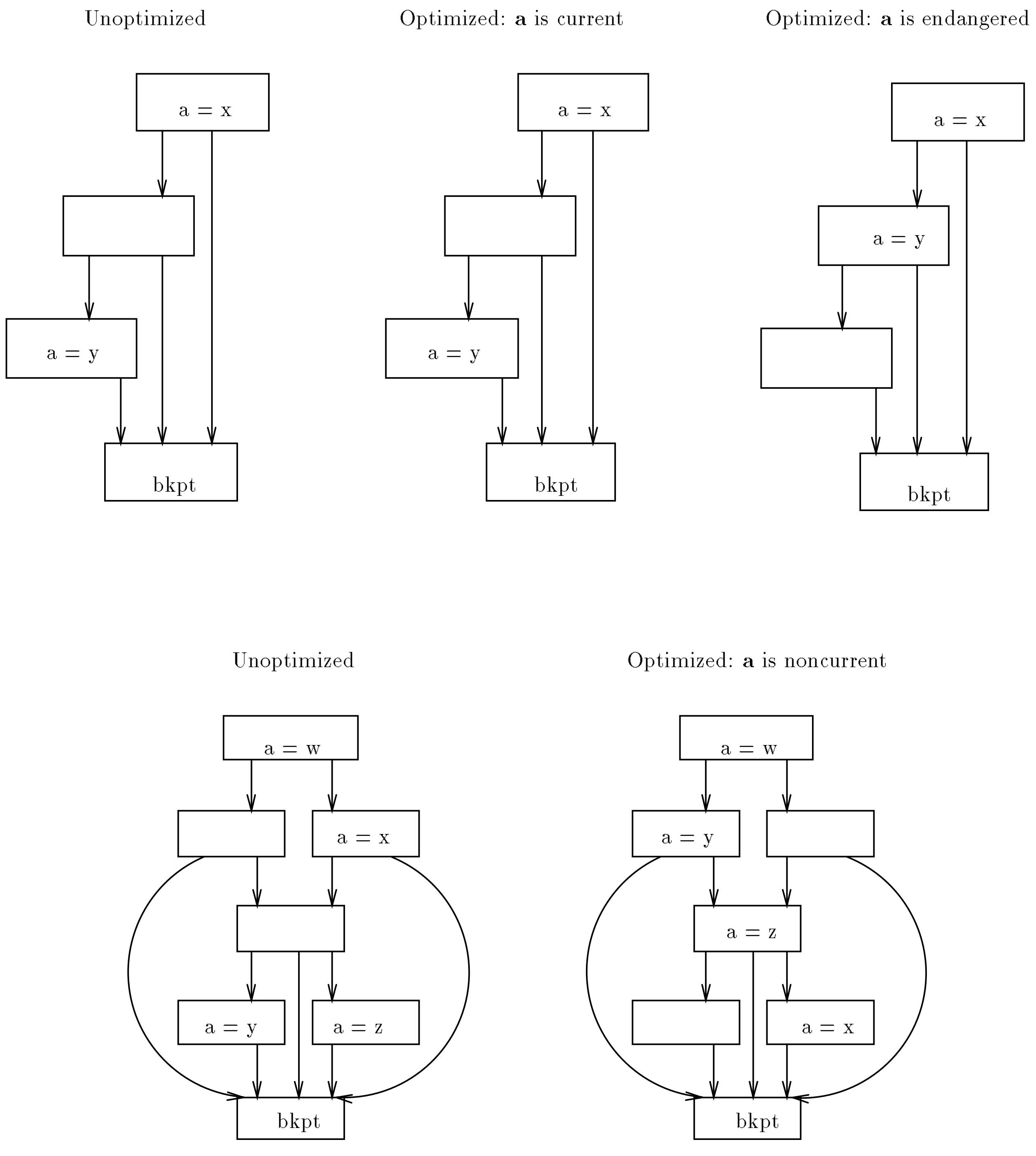


Figure 4.3: Stores that Reach **bkpt** are Exactly Those Generated from Definitions that Reach **bkpt**

costly, so it may be preferable to use an approximation to V's currentness at B that sacrifices accuracy for ease of computation. Such an approximation should be conservative – it may occasionally incorrectly tell you V is endangered, but it should never tell you that V is current when V in fact is not.

There is such an approximation, which, if the compiler saves the appropriate information, is simple to compute. The approximation is: *If no relevant code motion has occurred, V is current at B. If such motion is found, V may be conservatively claimed to be endangered at B.* Relevant code motion is any motion across block boundaries of stores generated from definitions that reach B.

Optimization can modify the shape of the flow graph, introducing or deleting node and edges. What does motion across block boundaries mean when block boundaries are fluid? Block boundaries can be fixed by the use of markers that are never moved by optimization. A marker is placed in the code stream at the end of each block. Since optimization never moves these markers, in the optimized code they denote the boundaries of blocks as they existed in the unoptimized code.[21] If the markers are uniquely identified, it is possible to determine which block contains a definition and which block contains the store generated from it; essentially, this is a method of embedding the pre-optimization block boundaries into the post-optimization flow graph.

It is not known how good this approximation is. However, because no code motion occurs in the absence of optimization, this approximation works perfectly on unoptimized code. Furthermore, to get to the inaccurate case there must be

- more than one definition of V reaching the breakpoint,
- more than one store into V reaching the breakpoint,
- stores that reach must be precisely the stores generated from the definitions that reach, and
- optimization involving code motion across a block boundary of a reaching store must have occurred. 1.5mm]

If, in this case, a conservative response is not deemed sufficient, the graph examinations described in Section 4.4 can be performed. Table 4.2 summarizes how to determine V's currentness when multiple definitions and multiple stores reach the breakpoint.

4.4 When All Else Fails

Let us examine the case in which comparing reaching sets does not give us an answer and relevant code motion has occurred. We are now assuming the conditions enumerated above.

In general, V is current at B if every path to B that goes through a definition of V also goes through the store into V generated from that definition, and neither the definition nor store are subsequently killed. The embedding of the pre-optimization block boundaries in the post-optimization flow graph from Section 4.3 allows us to proceed as if we have stores and definitions in a single graph.

[21] If a block is eliminated, its marker will be eliminated as well. This poses no problem, as we look to see what block a store ended up in, not what ended up in a particular block.

	Many definitions reach B
Many stores reach B	Were any of the stores generated from any of the definitions? No: noncurrent Yes: Were the stores exactly those generated from the definitions, and did every definition generate a store? No: endangered Yes: Was there any relevant code motion? No: current Yes: approximate with endangered, or perform graph examination

Table 4.2: The Many-Many Case

V is current at B iff for all definition/store pairs d,s where d defines V and s was generated from d the following hold:

1. If s has been moved DOWN out of the block containing d then
   (a) there is no path to s that did not go through d, and
   (b) for all paths from d to B along which d reaches B, s reaches B.
2. If s has been moved UP out of the block containing d then
   (a) there is no path to d that did not go through s, and
   (b) for all paths from s to B along which s reaches B, d reaches B.

Notice that case 2 above is identical to case 1 with the roles of d and s reversed.

Figure 4.4 attempts to capture the restrictions pictorially on an example in which the store has moved down. In the figure, d represents a definition of V and s represents the store generated from it (similarly for the primed versions).

Let the block containing `bkpt` be called Bottom. Let the block containing whichever of d and s is further from Bottom (ignoring any back edges in the graphs) be called Top, and the other be called Middle. Then we can state the conditions as

(a) there is no path to Middle that did not go through Top, and

(b) for all paths from Top to Bottom along which Top reaches Bottom, Middle reaches Bottom.

This in turn is equivalent to

(a) every path from the source of the flow graph to Middle passes through Top, and

(a) every path from Top to Bottom passes through Middle or through a block in which Top is killed.

Condition (a) can be tested by removing Top from the graph and determining whether there is a path from the source block to Middle (using a standard graph technique such as breadth-first search).

Condition (b) can be tested by removing Middle and all blocks in which Top is killed from the graph and determining whether there is a path from Top to Bottom. Definitions of V are killed by other definitions of V, and stores into V are killed by other stores into V. Note that, assuming Top contains a definition of V,

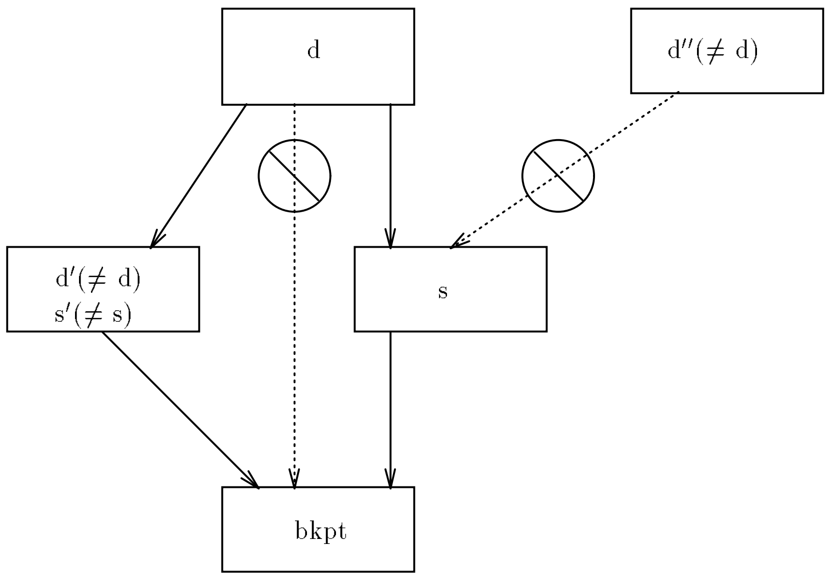


Figure 4.4: Paths if V is Current. Definitions of V are represents by d, d', and $d'\prime$; s and s' represent stores respectively generated from them.

removing every other block containing a definition of V will give the same results as removing just those on paths from Top to Bottom.

Thus V's currentness at B can be precisely determined in all circumstances, but in some cases an examination of the flow graph must be made.

5 Proof of Correctness

This section offers proofs of correctness of the case analysis summarized in Tables 4.1 and 4.2, with the exception of the graph examination.

5.1 Notation

d denotes a definition and s denotes a store. D denotes the set of definitions of a variable V that reach a breakpoint B, and S denotes the set of stores into V that reach B.

An equality test cannot compare a definition with a store, because they are two different types of entities. The operator $\tilde{=}$, which represents the *generates* or *generated from* relation, is used to compare definitions with stores:

Definition 12: $d \tilde{=} s$ means s was generated from d.[22]

The domain of $\tilde{=}$ extends to sets of definitions and stores:

Definition 13: $D \tilde{=} S$ means that $|D| = |S|$ and each $s \in S$ was generated from a distinct $d \in D$.

$\tilde{\in}$ maps a definition to the store it generates or a store to the definition it was generated from and then assert set membership:

Definition 14: $d \tilde{\in} S$ means the store generated from d is in S and $s \tilde{\in} D$ means s was generated from a definition in D.

Finally, $\tilde{\cap}$ is used to describe the two sets that result from the intersection of a set of definitions with a set of stores:

Definition 15: $S \tilde{\cap} D = S', D'$ which are maximal sets such that $S' \subseteq S$, $D' \subseteq D$, and $D' \tilde{=} S'$.

Clearly $|S'| = |D'|$, so *S' and D' are empty* is written $S \tilde{\cap} D = \emptyset$ (no $s \in S$ was generated from any $d \in D$).

5.2 Correctness

Table 5.1 is a combination of Tables 4.1 and 4.2 using the notation defined in this section. This section provides proofs of the assertions in the table, excepting the case requiring graph examination.

Theorem 1: *Table 5.1 correctly determines the currentness of a variable V at a breakpoint B, excepting the case requiring graph examination.*

The four entries in Table 5.1 are mutually exclusive and exhaustive. It suffices to prove that each entry is correct. The proof is by case analysis where each case corresponds to an entry in Table 5.1. In the proof, the cases are distinguished by the cardinalities of the reaching sets D and S.

[22] In an implementation, we suggest representing a definition and the store generated from it as one unit, which allows equality to be used to compare a definition with a store.

		Definitions of V that reach B	
		One (d)	Many (D)
Stores into V that reach B	One (s)	$d\tilde{=}s$? Yes: current No: noncurrent	$s\tilde{\in}D$? Yes: endangered No: noncurrent
	Many (S)	$d\tilde{\in}S$? Yes: endangered No: noncurrent	$S\tilde{\cap}D = \emptyset$? Yes: noncurrent No: $S\tilde{=}D$? No: endangered Yes: Was there relevant code motion? No: current Yes: approximate with endangered, or perform graph examination.

Table 5.1: The Cases Revisited

Proof 1:

1. $|D| = |S| = 1$

   Let $D = \{d\}$ and $S = \{s\}$. By the definitions of D and S, d reaches B along all paths to B, and s reaches B along all paths to B.

   (a) $d\tilde{=}s$: V is current at B by Definition 8.

   (b) $d\tilde{\neq}s$: V is noncurrent at B by Definition 9.

2. $|D| = 1$ and $|S| > 1$

   Let $D = \{d\}$ and let s be the store into V such that $d\tilde{=}s$.

   (a) $s \notin S$

   $\Rightarrow \nexists p$ a path along which s reaches B. V is noncurrent at B by Definition 9.

   (b) $s \in S$

   $\Rightarrow \exists p$ along which s reaches B. d reaches B along p. V is not noncurrent at B by Definition 9.

   $|S| > 1 \;\Rightarrow\; \exists s' \in S,\ s' \neq s \;\Rightarrow\; d\tilde{\neq}s'$. Let p be a path along which s' reaches B. p exists by definition of S. d reaches B along all paths, and thus along p. V is endangered at B by Definition 10.

3. $|S| = 1$ and $|D| > 1$

   This case is identical to case 2 with the roles of S and D reversed.

4. $|S| > 1$ and $|D| > 1$

   This case is further divided by how well the definitions in D match the stores in S:

   (a) $S\tilde{\cap}D = \emptyset$ (No store in S is generated from a definition in D)

   $\Rightarrow \forall s \in S,\ d \in D,\ d\tilde{\neq}s \;\Rightarrow \nexists p$ along which $d_p \in D$ reaches B, $s_p \in S$ reaches B, and $d_p\tilde{=}s_p$. V is noncurrent at B by Definition 9.

(b) $S \tilde{\cap} D \neq \emptyset$ and $S \tilde{\neq} D$

(At least one store in S is generated from a definition in D, but not all stores in S and definitions in D can be paired such that the store is generated from the definition)

$S \tilde{\neq} D \;\Rightarrow$ either

i. $\exists s \in S$ such that $\forall d \in D$, $d \tilde{\neq} s$

Let p be a path along which s reaches B, and let $d_p \in D$ be the definition that reaches B along p. $d_p \tilde{\neq} s \;\Rightarrow\; V$ is endangered at B by Definition 10. Or

ii. $\exists d \in D$ such that $\forall s \in S$, $d \tilde{\neq} s$

Let p be a path along which d reaches B, and let $s_p \in S$ be the store that reaches B along p. $d \tilde{\neq} s_p \;\Rightarrow\; V$ is endangered at B by Definition 10.

Furthermore, $S \tilde{\cap} D \neq \emptyset \;\Rightarrow\; \exists s \in S$, $d \in D$ such that $d \tilde{=} s$. The stronger claim that V is noncurrent at B may not hold because there may be a path along which both s and d reach B. Figure 5.1 is an example of such a situation. The claim that V is not noncurrent at B may also not hold: there may be no path along which both s and d reach B. Figure 5.2 is an example of such a situation.

(c) $D \tilde{=} S$

We have seen in Figure 4.3 that in this case V may be current, endangered, or noncurrent at B. In the absence of relevant code motion, V is current at B. That is, if $D \tilde{=} S$ and no store into V that reaches B has been moved out of the basic block containing the definition of V from which that store was generated, V is current at B.

Assume $D \tilde{=} S$ and no relevant code motion has occurred. Assume further that V is endangered at B. There must be some path p to B along which V is noncurrent. Let $d \in D$ be the definition that reaches B along p. $\exists s \in S$ such that $d \tilde{=} s$. p comprises a sequence of blocks $b_0, b_1, \ldots, b_n$ with d in b_i and B in b_n. By assumption, s is in b_i.

i. $b_i = b_n$

Since s and B are in the same basic block, there is no other path by which s can reach B. Since $D \tilde{=} S$, s reaches $B \;\Rightarrow\; s$ reaches B along $p \;\Rightarrow\; V$ is current along p by Definition 6, a contradiction.

ii. $b_i \neq b_n$

s and B are in distinct basic blocks. s must reach the exit of b_i because if it did not, it could not reach B along any path, yet $s \in S$ implies that it does reach B. V is noncurrent at B along $p \;\Rightarrow\; \exists s' \in S$ such that s' reaches B along p, $s' \neq s$, and s' kills s along p. s' must be in some block b_j along p, $i < j <= n$. Since $D \tilde{=} S$, $\exists d' \in D$ such that $d' \tilde{=} s'$. However, d' cannot be in b_j or it would have killed d along p, but by assumption d reaches B along p. Therefore s' has been moved out of the basic block containing d', a contradiction.

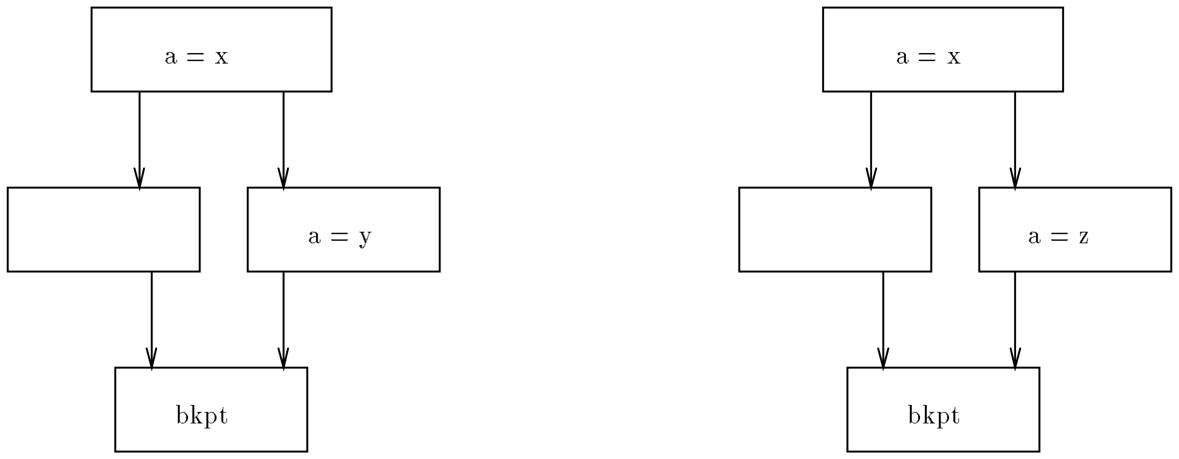


Figure 5.1: $S \tilde{\cap} D \neq \emptyset$ and $S \tilde{\neq} D$ and **a** is Current at **bkpt** along the Leftmost Path

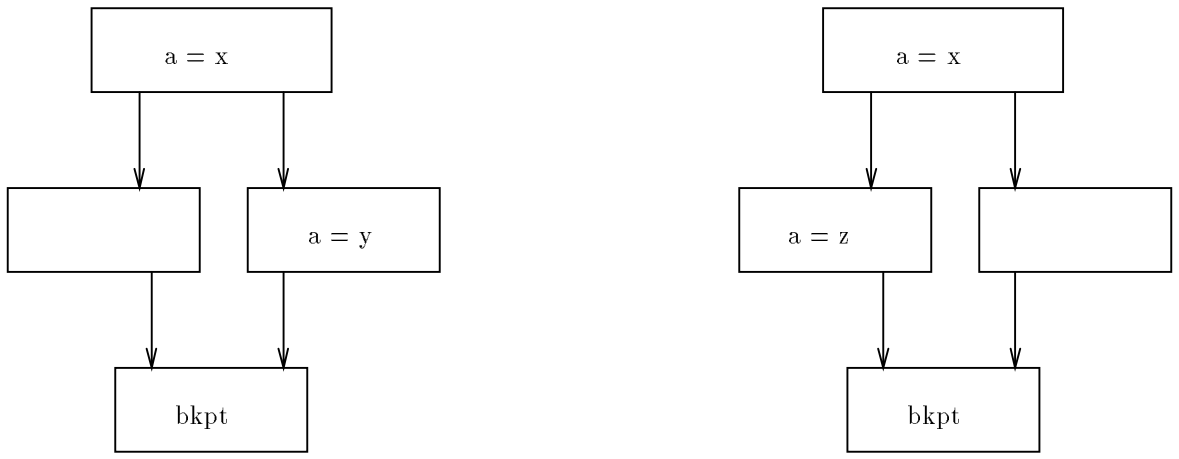


Figure 5.2: $S \tilde{\cap} D \neq \emptyset$ and $S \tilde{\neq} D$ and **a** is Noncurrent at **bkpt**

6 When a Breakpoint has Moved

Semantic breakpoints introduce additional complexity into currentness determination. This section merely outlines the difficulties. For a more complete discussion of currentness at semantic breakpoints, see [Cop92].

When a semantic breakpoint is reached, the point in the optimized code at which execution is suspended (and the user examines a variable's actual value) may not correspond to the point at which the user expects execution to be suspended (the point at which the user intended to examine the value). There are four distinct situations that 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 currentness 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 a source of unexpected behavior, but no additional work is required for currentness determination because the user never gets to ask for the value of a variable at the semantic breakpoint.
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.

In situations 2 and 4 we need to be able to compare the actual value of a variable at a representative instruction R (the semantic breakpoint, where the user examines the value) with its expected value at a representative instruction $R' \neq R$ (the syntactic breakpoint, where the user expects to be examining the value). Our current definitions of current, noncurrent, and endangered do not cover these situations.

There is a further problem. Consider Figure 1.1. For `bkpt` to be reached in the optimized code, the right-hand paths must be taken. If the unoptimized code is run on the same inputs, the right-hand paths will be taken, so optimization does not affect the value `a` will have at the semantic breakpoint for `bkpt`: `a` is current at `bkpt`. However, the reaching sets are $D = \{\mathtt{a} = \mathtt{x}, \mathtt{a} = \mathtt{y}, \mathtt{a} = \mathtt{z}\}$ and $S = \{\mathtt{a} = \mathtt{x}, \mathtt{a} = \mathtt{y}\}$. Comparing the reaching sets according to Table 5.1 gives the conclusion that `a` is endangered at `bkpt`. Thus the reaching set comparison that is adequate for syntactic breakpoints is inadequate for semantic breakpoints.

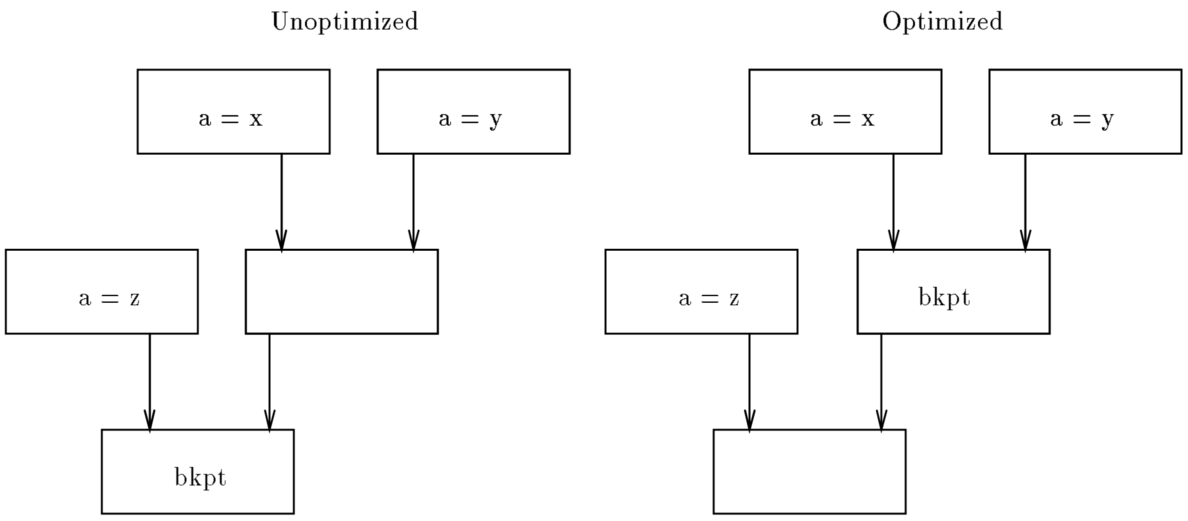


Figure 6.1: Oddly enough, **a** is current at **bkpt**

7 Reaching-Definitions Support

The tool that computes the sets of definitions and stores that are needed to compute a variable's currentness needs information about the definitions and stores and control flow of the program. If the compiler is the tool that performs these computations, existing compiler data structures can be modified for the task. In order to abstract the information needed solely for this task, this section assumes that the compiler will provide the necessary information to the debugger, and the debugger will perform the reaching-definitions computation.

7.1 Debug Records

The compiler provides the debugger with information about every declaration and statement in the program. The collection of information about a statement (declaration) is called a debug record. Each debug record for a statement represents a breakpoint location. Each debug record for an assignment represents (in addition to the breakpoint location) a definition and the store generated from it. A distinct debug record is produced for each modification to each program variable, so more than one debug record is produced for a statement that has side effects.[23] For example, the following code causes 6 debug records to be produced:

`int a, b, c;`	(Produces three declaration debug records.)
`a = 0;`	(Produces one statement debug record.)
`b = c++;`	(Produces two statement debug records: one for the assignment into b, and one for the side effect on c.)

A debug record R for a statement S has the following fields:

- *Var(R)* — a variable name,
- *Sref(R)* — a source reference,
- *Cref(R)* — a code reference,
- *Moved(R)* — a flag, and
- *Equiv(R)* — an equivalence class identifier.

The Var field identifies the variable assigned into by S. If S does not assign into a variable, the Var field is null. The Sref field contains the source reference for S (file name and line number, and perhaps which statement on the line, if the debugger is to handle lines with multiple statements). The Cref field contains the address of the representative instruction for S (the breakpoint location for S).[24] If no instruction is generated for S, the Cref field is null, unless the debug record describes a declaration, in which case the Cref field contains the address of the instruction that allocates storage for the declared variable.[25] The Moved

[23] More than one debug record is produced for a statement that has more than one location at which user-visible changes occur. This is true of statements with side effects. It is also true of many loop constructs. A C `for` loop may have three places of interest to the user corresponding to its three expressions, and each needs a debug record if the debugger is to be able to break at each one.

[24] The Cref is *not* the address of the storage location for a variable (a data address), but rather the address of the representative instruction of a statement (a code address).

[25] The Cref for the declaration of a static variable contains the start address of the program.

field encodes whether the code for S has been moved, and if so, whether it has been moved out of the basic block in which it originated. The *Equiv(R)* field records the equivalence class that a definition (Sref) and store (Cref) fall into.[26]

7.2 Flow Graphs

The compiler also provides the debugger with two representations of the control flow of the program.

A flow graph representing the basic block structure before optimization is called the *source graph*. Each node in the source graph corresponds to a basic block and contains a sequence of (pointers to) debug records[27], one for each statement and side effect within the block in the order in which they occur in the unoptimized code.

A flow graph representing the basic block structure after optimization is called the *object graph*. Each node in the object graph corresponds to a basic block and contains a sequence of (pointers to) debug records that corresponds to the sequence of statements and side effects that have ended up in that block. The basic block structure prior to optimization is embedded in the object graph through the use of markers that are never moved by optimization. A marker is placed in the code stream at the end of each block in the source graph before optimization is performed. The object graph is a copy of the source graph on which optimizations have been tracked. In the object graph the markers denote the pre-optimization block boundaries.

Control flow information can be used by a debugger for purposes other than currentness determination. For example, statement stepping (often called source-line stepping) is one of the more difficult capabilities to implement because it is difficult to determine where the next breakpoint(s) should be set. With control flow information, this problem becomes simple. Using the program flow graphs and debug records described in this section, the current breakpoint is at the Cref of the debug record R for the current statement. If R is not the last record in its block in the object graph, the next breakpoint can be set at the Cref of the next record. If R is the last record in its block, breakpoints can be set at the Crefs of the first record of each successor block.

7.3 Reaching Definitions

The flow graphs are used to compute reaching definitions. We are interested in determining, for each statement that defines a variable V and reaches a breakpoint B in the the unoptimized code, whether its corresponding object code reaches B. Both statement and breakpoint locations are represented with debug records, so the desired determination can be made by computing which debug records representing definitions of V (or stores into V) reach the debug record representing the breakpoint B.

[26]If the compiler has determined that a set of definitions represents the same computation, all of the stores generated from those definitions represent the same computation, thus the debug record, which represents both a definition and a store, needs only a single field to represent both the equivalence class that the definition falls into and the equivalence class that the store falls into.

[27]There is a single set of debug records that is shared between the two flow graphs, however, for all intents and purposes the nodes are treated as if they contain debug records as opposed to pointers to records.

These reaching definitions are computed across, as well as within, basic blocks, so those records that must reach B (such as definitions occurring prior to B in the same block) can be distinguished from records that may reach B (definitions occurring on some but not all paths to B). The set of definitions that may reach B is computed based on the Sref field of the debug records in the source graph. The set of stores that may reach B is computed based on the Cref field of the debug records in the object graph.

The beginning of the program or subroutine (the start node of a connected component of a flow graph) constitutes a null definition and store of each variable. This ensures that some definition (store) for each variable reaches B along every path. This also ensures that if only one definition (store) for a variable reaches B, it reaches along all paths to B.

In the absence of pointers and array references, reaching definitions could be computed using a standard algorithm (Aho and Ullman [AU77]). This would produce at most one definition of a given variable at the exit of a block. Using such an algorithm, an assignment through a pointer or array reference would kill all pending definitions. This would destroy information required for currentness determination. In the presence of pointers and array references, reaching definitions must be computed using a modified algorithm in which an assignment through a pointer or array reference does not kill previous definitions, thus more than one definition of a given variable may reach any point, including the exit of a block.

7.4 Semantic and Syntactic Breakpoint Locations

Under the representation described in Sections 7.1 and 7.2, the semantic breakpoint location for a statement is the Cref of the debug record for that statement.

The syntactic breakpoint location L for a statement S is determined as follows:

```
If the representative instruction for $S$ has not been moved
    $L$ is the address of that instruction.
If that instruction has been moved
    if the block that originally contained it does not appear at all in the optimized code,
        $L$ is undefined,
    else if any representative instructions for statements following $S$ within the block containing $S$
    have not been moved,
        $L$ is the location of the first of these,
    else $L$ is the location of the last representative instruction within the block containing $S$.
```

8 Summary

It is not always possible to completely debug an unoptimized version of a program. Examples have been given in which optimization changes the behavior of a program even when the optimizer is correct. This is not a new result, but such examples have not previously appeared in the literature.

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 paper has described a mapping between statements and breakpoints that provides a reasonable approximation to what the naive user would expect when 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 is 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.

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 currentness of a variable if it is to provide truthful behavior on optimized code. Hennessy [Hen82] [CM91a] and Coutant et al [CMR88] give solutions to special cases of the currentness determination problem. Table 8.1 summarizes a general solution to the problem for sequential optimizations. These results hold in the presence of both local and global optimizations and require no information about which optimizations have been performed.

This paper has described the information a compiler must make available to the debugger for this task, as well as the nature of the information the debugger can provide to the debugger user when the user asks for the value of an endangered variable.

For most optimizations, the results described in this paper are precise (i.e., a variable claimed to be current is current, a variable claimed to be endangered is endangered, etc.) except 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.[28]

For some optimizations, the results may be conservative. These optimizations are those that duplicate code in such a manner that the duplicates are not in the same equivalence class (two duplicates do not represent equivalent computations, as in loop unrolling).[29] Table 8.2 lists representative optimizations and shows whether the results are precise or conservative on them.

The method to precisely determine a variable's currentness in the most difficult case may be expensive (see Section 4.4). Section 4.3 describes an inexpensive conservative approximation to the precise result in

[28] An infeasible path is one that cannot be taken in any execution.

[29] Strictly speaking, we have no results for such optimizations, as Definitions 1, 4 and 5 are not strong enough to cover such optimizations. However, given a duplicated store s, assuming $s \tilde{\notin} D$ gives reasonable but conservative results.

	One definition, d, reaches B	Many definitions reach B
One store, s, reaches B	Was s generated from d?	Was s generated from one of the definitions that reach?
	Yes: current	Yes: endangered
	No: noncurrent	No: noncurrent
Many stores reach B	Was one of the stores generated from d?	Were any of the stores generated from any of the definitions?
	Yes: endangered	No: noncurrent
	No: noncurrent	Yes: Were the stores exactly those generated from the definitions, and did every definition generate a store?
		No: endangered
		Yes: Was there any relevant code motion?
		No: current
		Yes: approximate with endangered, or perform graph examination

Table 8.1: The Four Cases

Optimization	Algorithm Accuracy
common subexpression elimination	Generally Precise
cross-jumping	Generally Precise
instruction scheduling	Generally Precise
other code motion	Generally Precise
partial redundancy elimination	Generally Precise
loop reordering	Generally Precise
induction-variable elimination	Generally Precise
loop fusion	Generally Precise
loop unrolling	Conservative
inlining (procedure integration)	Conservative

Table 8.2: Precision of Results on Representative Optimizations — *Generally Precise* means precise except when a variable is current along all feasible paths but noncurrent along some infeasible path.

this case.

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 currentness 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 `17`; `y` was set at line 506 to `y+bar(w)`.'. This was called *flowback analysis* by Balzer [Bal69], and has been investigated by others ([MC91], [Kor88]); reaching sets may be adaptable to this purpose.

Another research direction is dynamic currentness 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. And finally, an exciting possibility is extending the breakpoint model and currentness determination techniques to parallel code, where noncurrent variables are common.

References

[AU77] A. V. Aho, J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, Menlo Park, CA, 1977.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers Principles, Techniques, and Tools," Addison-Wesley, Menlo Park, CA, 1986.

[Bal69] R. M. Balzer, "EXDAMS - EXtendable Debugging and Monitoring System," *Proceedings of AFIPS Spring Joint Computer Conference*, Vol 34 pp. 125-134, 1969.

[CM91a] M. Copperman, C. E. McDowell, "A Further Note on Hennessy's "Symbolic Debugging of Optimized Code", UCSC Technical Report UCSC-CRL-91-04, February 1991. Submitted for publication to *ACM Transactions on Programming Languages and Systems*

[CM91b] M. Copperman, C. E. McDowell, "Debugging Optimized Code Without Surprises," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, November 1991.

[Cop90] M. Copperman, "Source-Level Debugging of Optimized Code: Detecting Unexpected Data Values," University of California, Santa Cruz technical report UCSC-CRL-90-23, May 1990.

[Cop92] M. Copperman, "Source-Level Debugging of Optimized Code Without Surprises," Doctoral thesis, unpublished draft from University of California, Santa Cruz, 1992.

[CMR88] D. Coutant, S. Meloy, M. Ruscetta "DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

[FM80] P. H. Feiler, R. Medina-Mora, "An Incremental Programming Environment," Carnegie Mellon University Computer Science Department Report, April 1980.

[Hen82] J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, 1982.

[Kor88] B. Korel, "PELAS Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253-1260, September 1988.

[MC88] B. Miller, J. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 125-134, Madison, Wisconsin, 1988.

[MC91] B. Miller, J. Choi, "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 491-530, 1991.

[Pic90] D. Pickens, MetaWare Incorporated, Santa Cruz, CA, personal communication regarding the MetaWare High C compiler.

[PS88] L. L. Pollack, M. L. Soffa, "High Level Debugging with the Aid of an Incremental Optimizer," *Hawaii International Conference on System Sciences*, January 1988.

[PS92] L. L. Pollack, M. L. Soffa, "Incremental Global Reoptimization of Programs," Draft from Department of Computer Science, University of Pittsburgh, May 1991. To appear in *ACM Transactions on Programming Languages and Systems* in 1992.

[Str91] L. Streepy, "CXdb A New View On Optimization," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, November 1991.

[WS78] H. S. Warren, Jr., H. P. Schlaeppi, "Design of the FDS interactive debugging system," IBM Research Report RC7214, IBM Yorktown Heights, July 1978.

[Ze83a] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," *Research Report CSL-83-1* , Xerox Palo Alto Research Center, Palo Alto, CA, Jan. 1983.

[Ze83b] P. Zellweger, "An Interactive High-Level Debugger for Control-Flow Optimized Programs," *SIGPLAN Notices*, Vol. 18, No. 8, pp. 159-172 Aug. 1983.

[Zel84] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Research Report CSL-84-5, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.

[ZJ90] L. W. Zurawski, R. E. Johnson, "Debugging Optimized Code With Expected Behavior," Unpublished draft from University of Illinois at Urbana-Champaign Department of Computer Science, August 1990.