A New Breakpoint Implementation Scheme for Debugging Globally Optimized Code

Le-Chun Wu* Wen-mei W. Hwu†

*Department of Computer Science
†Department of Electrical and Computer Engineering and
The Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801
Email: {lcwu, hwu}@crhc.uiuc.edu

Technical Report IMPACT-98-06

July 1998

Abstract

With an increasing number of executable binaries generated by optimizing compilers in today's high-performance computing, providing a clear and correct source-level debugger for programmers to debug optimized code has become a necessity. Implementing source breakpoints is a fundamental aspect of such a debugger. In this paper, a new source breakpoint implementation scheme which consists of a new code location mapping scheme and a forward recovery model is proposed. The approach is aimed at solving the fundamental problems suffered by traditional methods. By taking over the control early and executing instructions under our forward recovery model, the debugger can preserve and gather the required program states. With this information accumulated and the help of a data location tracking method, the expected values of user variables can be recovered at source breakpoints. The new code location mapping scheme helps the debugger to determine where to suspend and resume the normal execution and decide if a source breakpoint should be reported. The algorithms and theoretical foundations for constructing and calculating the mappings are presented.

1 Introduction

In today's high-performance computing, compilers are playing an increasingly important role by optimizing programs to fully utilize advanced architecture features. With an increasing number of executable binaries being highly optimized, it has become a necessity to provide a clear and correct source-level debugger for programmers to debug optimized code.

In general, there are two ways for an optimized code debugger to present meaningful information of the debugged program [1]. It provides *expected behavior* of the program if it hides the optimization from the user and presents the program behavior consistent with what the user expects from the unoptimized code. It provides *truthful behavior* if it makes the user aware of the effects of optimizations and warns of surprising outcomes when the expected answers to the debugging queries cannot be provided. Although it is not always possible to recover the program behavior to what the user expects without constraining the optimization performed or inserting some instrumentation code [2], it is desirable for the user to see as much expected program behavior as possible. Therefore this paper will focus on a framework for recovering expected behavior.

One of the most frequently used functionalities of a source-level debugger is for the user to set breakpoints and examine variables' values at these points. Therefore accurate implementation of source breakpoints is a basic requirement in supporting optimized code debugging. Traditionally, the debugger maps a breakpoint set at a source statement to an object code location, called *object breakpoint* [3], and the execution halts when this object location is reached. The debugger then uses the program state at this point to answer the user's inquiries. However, compiler optimizations cause difficulties to the debuggers using this traditional approach. For example, Figure 1 shows a C source program and its assembly code optimized by scheduling and register allocation. We assume that the debugger uses a mapping that would set the object breakpoint at $I5$ when a source breakpoint is set at statement $S2$. When the debugger halts the execution at $I5$, instructions from statement $S3$ ($I3$ and $I4$) have been executed, which causes variable y to be updated prematurely. Also instruction $I6$ which should have been executed in order to obtain the expected value of variable a has not been executed at this point.

There are two primary aspects associated with code optimization that cause the traditional scheme difficulties in handling optimized code. First, optimization complicates the mapping between the source code and the object code. Due to code duplication, elimination, and re-ordering

```
breakpoint --> S2:  x = 2;
```

```
                S1:  a = b + c;
breakpoint -->  S2:  x = 2;
                S3:  y = z * 3;
```

(a)

```
I1:  ld   r1, b        <S1>
I2:  ld   r2, c        <S1>
I3:  ld   r5, z        <S3>
I4:  mul  r6, r5, 3    <S3>
I5:  mov  r4, 2        <S2>
I6:  add  r3, r1, r2   <S1>
```

(b)

Figure 1: An example program (a) C-style source code (b) Assembly code

caused by optimization, it can be hard for the debugger to identify the object breakpoint location when the user sets a source breakpoint. Second, only the program state of a single point is available once the execution is halted by the debugger. Therefore, optimized code debuggers that adopt a traditional breakpoint implementation scheme usually have problems reporting the expected values of the variables which are updated either too early or too late [2, 4].

In this paper, we propose a new breakpoint implementation scheme which overcomes the problems encountered by the traditional method. Our approach is motivated by the observation that in order for the debugger to provide the expected variable values, the program states changed by the out-of-original-source-order instructions have to be tracked by the debugger. To do this for a breakpoint at source statement S, the debugger suspends execution before executing any instruction that should happen after S. The object code location where the debugger suspends the normal execution is referred to as the *interception point*. It then moves forward in the instruction stream executing instructions (basically single-stepping through the instructions) using a new *forward recovery* technique which keeps track of program states. When the debugger reaches the farthest extent of instructions which should happen before S, referred to as the *finish point*, it begins to answer the user's inquiries. When reporting the value of a variable, it uses the preserved program states to recover the expected values.

The basic idea of our approach can be illustrated by the example in Figure 1. If the user sets a breakpoint at source line $S2$, since instruction $I3$ originates from source line $S3$, the debugger suspends execution at $I3$. The debugger keeps executing instructions using the forward recovery technique until instruction $I6$ is executed because it originates from source line $S1$ which should be executed before the breakpoint. The debugger then hands over the control to the user and starts taking user's requests. During forward recovery execution, the original contents of the registers which are updated prematurely are preserved to provide the user with the expected variable values

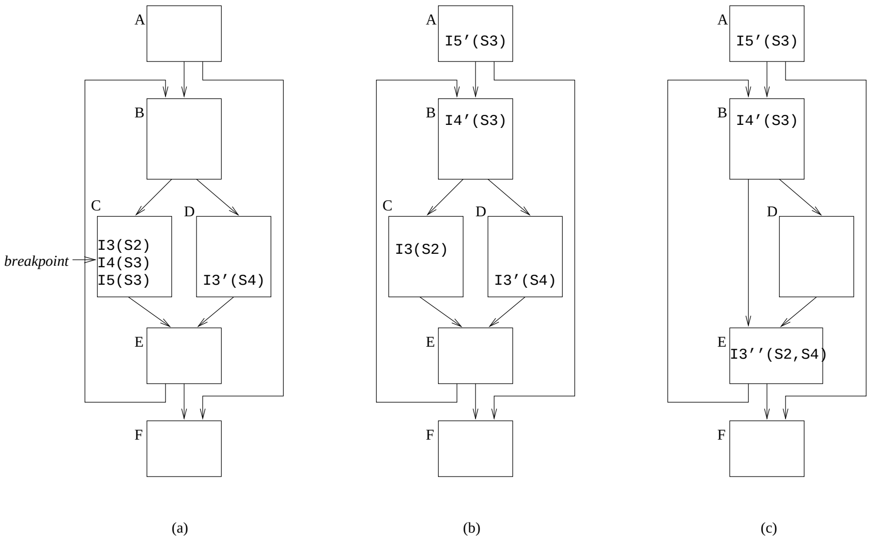


Figure 2: A control flow graph example (a) original program (b) after code hoisting (c) after tail merging

at $S2$.

The basic idea of our breakpoint implementation strategy looks straightforward for a straight-line code example shown in Figure 1. However, the scheme must address several challenging issues when dealing with globally optimized code. We use the example shown in Figure 2 to illustrate how global optimization complicates our scheme. Figure 2 (a) shows the control flow graph of an unoptimized program where instruction $I3$ is from statement $S2$, $I4$ and $I5$ are from $S3$, and $I3'$ is from $S4$. Basic block C contains only $I3$, $I4$, and $I5$. Figure 2 (b) shows an optimized version of the program where instruction $I5$ is moved out of loop and instruction $I4$ is hoisted to basic block B. Figure 2 (c) shows an optimized version of the program where $I3$ and $I3'$ are merged and sunk to basic block E. Basic block C is therefore left empty. Suppose a source breakpoint is set at statement $S3$ by the user. The problems which need to be addressed by our scheme include:

1. How to calculate all the possible interception points and finish points?

   With code being re-ordered globally, instructions which should be executed after a source breakpoint might be hoisted above the breakpoint on different paths leading to the breakpoint. For example, in Figure 2 (b), instruction $I4'$ and $I5'$ are the instructions which should be executed after the breakpoint but were hoisted. We can see that for the first iteration of the loop, the debugger should suspend the execution at $I5'$, while for the rest of the loop iterations the debugger should suspend the executed at $I4'$. Therefore $I5'$ and $I4'$ should both be interception points of $S3$. Similarly, in Figure 2 (c), $I3$ should be executed before the breakpoint but was sunk to basic block E. The debugger needs to be able to identify where $I3''$ is and continues its forward recovery until $I3''$ is executed. Hence it is necessary to devise a set of systematic algorithms to calculate all the possible interception points and finish points.

2. How does the debugger confirm a source breakpoint?

   To preserve the required program states, the debugger has to suspend the execution early at the interception point. However, reaching an interception point of a source breakpoint does not necessarily mean the breakpoint should be reported to the user. Consider Figure 2 (b). After taking over control at instruction $I5'$, which is an interception point of $S3$, the debugger should report the breakpoint only when basic block C is reached. Otherwise, it should continue the normal execution without reporting the breakpoint. Apparently, for the debugger to be able to confirm the breakpoint, some mapping scheme which would map statement $S3$ to an object location inside basic block C has to be devised.

   Sometimes the compiler cannot even find a single object location in the optimized code to correctly map a source statement to. Figure 2 (c) illustrates such a case where basic block C is removed after optimization. We can see that there is no single object location which by itself can be used by the debugger to decide if statement $S3$ will be reached or not. Some branch conditions will need to be incorporated into the mapping scheme to help the confirmation of a breakpoint.

3. How does forward recovery work?

   For globally optimized code where instructions are moved across branches, it is important for the debugger to finish all the instructions that should be executed during forward recovery before reporting variable values to the user. Also the forward recovery technique should ensure that all the source breakpoints are reported to the user in the order consistent with what the user expects.

Our breakpoint implementation scheme consists of a new code location mapping scheme and a debugger forward recovery model. The code location mapping scheme enables the debugger to correctly determine where to suspend and resume the normal execution and decide if a source breakpoint should be reported. The forward recovery model ensures that the breakpoints and exceptions behave the way consistent with what the user expects. This new breakpoint scheme addresses the aforementioned problems caused by global optimization and provides a foundation for the design of an optimized code debugger. Working with a data location tracking method (such as the ones proposed in [4] and [5]) and some other necessary debugging information, the debuggers adopting our breakpoint implementation scheme is capable of recovering the expected program behavior.

The remainder of this paper is organized as follows: Section 2 discusses the new code location mapping scheme. Section 3 describes how our forward recovery model works. Section 4 discusses some of the previous works and compares our approach with them. Section 5 contains our conclusions.

2 Code location mapping

A debugger usually uses two kinds of code mappings [1, 2]: the *object-to-source* mapping which the debugger uses to report the faulty statement when an exception occurs, and the *source-to-object* mapping which the debugger uses to determine where to suspend the normal execution and decide if a source breakpoint should be reported. Since we are only interested in the implementation of user breakpoints in this paper, we will be focusing on source-to-object mapping in our scheme. The object-to-source mapping, nonetheless, can easily be built from the execution order information preserved during compilation (see Section 2.2.1).

Unlike the conventional source-to-object mapping scheme where a source statement is mapped to a single object location, our approach maps a statement to a set of object locations which can be classified into four categories with different functionalities: *anchor points*, *interception points*, *finish points*, and *escape points*. Interception points are the object locations where the debugger should suspend the normal execution and start forward recovery. Finish points are the object locations where the debugger should stop forward recovery and begin to take the user's requests. Escape points are used for the debugger to determine that a source breakpoint should not be reported. Anchor point information is the base for deriving interception, finish, and escape points, and needs to be constructed and maintained by the compiler. Interception points, finish points, and escape points can be derived from the anchor point information at either compile time or debug time. We will discuss each of these object locations in the following subsections.

2.1 Anchor points

In a traditional mapping scheme, a source breakpoint at statement S is mapped to a single object location (usually the first instruction of S). Without optimization, reaching this object location at run time means statement S is reached (providing the compiler is correct) and the debugger should report the breakpoint to the user.

Optimization, however, leaves this simple scheme insufficient. During optimization, the first instruction of a statement (or even the whole statement) might be deleted or moved away from its original place. Reaching the first instruction of a statement S does not necessarily mean S will be reached in the original source program. Sometimes the compiler cannot even find a single object location in the optimized code to correctly map a source statement to, as illustrated by the example shown in Figure 2 (c).

In order for the debugger to be able to correctly confirm a source breakpoint for globally optimized code, we extend the traditional simple source-to-object mapping by associating each source statement with *anchor point* information. An anchor point of a source statement is an object code location. Each anchor point comes with a boolean condition referred to as *anchoring condition*. When an anchor point of a source statement is reached during execution and its anchoring condition is true, the breakpoint set at that source statement should be reported.

Anchor point information for each source statement is constructed and maintained by the compiler. Before any optimization is performed, the anchor point of a source statement S is set to the first instruction of S and the anchoring condition is set to boolean value 1 (true).

During the process of code optimization, when code duplication optimization such as loop unrolling, function inlining, and loop peeling is performed, if an anchor point of statement S is contained in the duplicated code, the anchor point information is also duplicated. With the anchor point(s) of every duplicated instance of statement S associated with S, the debugger can correctly report the breakpoint set at S when any of S's instances is reached.

When an instruction I which is an anchor point of statement S is deleted or moved away from

step 1 If I has an immediate succeeding instruction J in the same basic block, J becomes an anchor point of S and the anchoring condition is boolean value 1.

step 2 else if I has an immediate preceding instruction J in the same basic block, J becomes an anchor point of S and the anchoring condition is boolean value 1.

step 3 else, all of I's immediate preceding instructions, $J_1, J_2, ..., J_k$ (where $k \geq 1$), become anchor points of S. If J_i is a conditional branch instruction, the condition under which J_i will branch to I becomes the anchoring condition. Otherwise, the anchoring condition is boolean value 1.

Figure 3: An algorithm for maintaining the anchor point information of statement S when instruction I, which is an anchor point of S, is being removed.

its original place, the compiler will modify the anchor point information of S using the algorithm shown in Figure 3.

Note that the algorithm in Figure 3 is based on the following assumptions: (1) Conditional branches will not be removed (assuming no predicated code). Thus any instruction I which is being removed is never a conditional branch and its anchoring condition is always 1. If the condition of a branch is a constant, our method allows the branch to be treated as an unconditional jump and thus allows it to be removed. (2) Every function in the object code has a section of prologue code which will not be removed. Thus any instruction which is being removed has an immediate predecessor.[1]

Figure 4 shows a control flow graph example where the whole basic block D is removed due to optimization. Based on our algorithm, both instruction $I1$ and $I2$ become the new anchor points of $S1$ with the anchoring conditions of boolean value 1 and $(r1 > 5)$ respectively.

After a series of optimizations is performed, a source statement might have more than one anchor points and an object instruction might be the anchor point of several source statements.

2.1.1 Proof of correctness

To prove that the anchor point information maintained by the compiler using the algorithm shown in Figure 3 is correct for the debugger to unambiguously decide if a source breakpoint should take effect or not, we need to first introduce the concept of *reaching condition*.

[1]If instruction I is in an unreachable block, I might not have a predecessor. However, the case is irrelevant because breakpoints set inside this block would never take effect anyway.

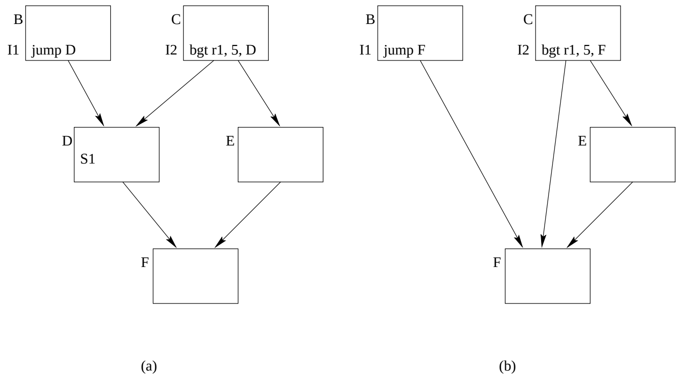


Figure 4: A control flow graph example (a) before removing basic block D (b) after removing basic block D.

Definition 1 *The reaching condition of an instruction I, RC_I, is a boolean expression comprising program variables and intermediate results so that when the condition is true, instruction I will be reached from the function entry point.*

Note that we assume that conditional branches will not be removed during optimization, therefore the reaching condition of an instruction remains the same during optimization as long as the instruction itself is not moved or deleted.

There might be more than one path which can lead to an instruction. For an instruction to be reached through a specific path, on this path every branch condition under which the path will be taken has to be true. Therefore, the *single-path reaching condition* of instruction I through a specific path P, $RC_{I,P}$, is the conjunction of every branch condition on P under which P is taken. That is,

Definition 2 $RC_{I,P} = \bigwedge_{i=1}^{n} C_i$*, where* C_i *is the condition of branch* i *under which* P *is taken, and* n *is the number of branches on* P*.*

Since an instruction can be reached through every path leading to it, the operational definition of the reaching condition of instruction I, RC_I, is the disjunction of all the I's single-path reaching conditions. That is,

Definition 3 $RC_I = \bigvee_{i=1}^{n} RC_{I,Pi}$*, where* Pi *is the* i*th path leading to* I *and* n *is the number of different paths leading to* I*.*

The reaching condition of an instruction can also be derived from that of its predecessors as the following theorem shows:

Theorem 1 *The reaching condition of the first instruction I of a basic block is equal to*

$$\bigvee_{i=1}^{n} RC_{Ji} \wedge BC_{Ji}$$

, where Ji is the ith immediate preceding instruction of I, RC_{Ji} is the reaching condition of instruction Ji, BC_{Ji} is the branch condition under which Ji will branch to I, and n is the number of I's immediate preceding instructions.

For any two instructions in the same basic block, when control reaches one instruction, it will definitely reach or have reached the other. Therefore,

Theorem 2 *All the instructions in the same basic block have the same reaching condition.*

As we mentioned earlier, a source breakpoint will be reported only when any one of its anchor point(s) is reached and the corresponding anchoring condition is true. The condition for the debugger to report a source breakpoint is referred to as *breakpoint confirmation condition.*

Definition 4 *The breakpoint confirmation condition of a source statement S, BCC_S, is*

$$\bigvee_{i=1}^{n} RC_{Ii} \wedge AC_{Ii}$$

, where Ii is the ith anchor point of S, AC_{Ii} is the anchoring condition of Ii, and n is the number of S's anchor point(s).

Before any optimization is performed, our scheme will map the anchor point of statement S to the first instruction, say I, of S and set the anchoring condition to 1. Assuming the compiler is correct, it is true that for the unoptimized code the breakpoint set at S should be reported if and only if I is reached. That is, before any optimization is performed, the breakpoint confirmation condition of S is a sufficient and necessary condition for the breakpoint set at S to be reported. Therefore if we can prove that the breakpoint confirmation conditions before and after the algorithm in Figure 3 is applied are the same, the algorithm is correct.

Theorem 3 *When an instruction I, which is an anchor point of source statement S, is removed due to optimization, the breakpoint confirmation conditions of S before and after the algorithm in Figure 3 is applied are the same.*

Proof: We prove the above theorem in two cases.

Case 1: Instruction I is the only anchor point of S. Therefore, $BCC_{S,before} = RC_I \wedge 1 = RC_I$.

When the algorithm in Figure 3 is applied, if either step 1 or step 2 is true, J becomes one and the only one anchor point of S and the anchoring condition is 1. Since J and I are in the same basic block, according to Theorem 2, $RC_J = RC_I$.

Thus, $BCC_{S,after} = RC_J \wedge 1 = RC_I \wedge 1 = BCC_{S,before}$.

If step 3 is applied, all the I's immediate preceding instructions, $J_1, J_2, ..., J_k$, become the anchor points of S, and the branch condition of J_i, BC_{Ji}, under which J_i will branch to I becomes the anchoring condition of J_i, AC_{Ji}.

Hence

$$\begin{aligned} BCC_{S,after} &= \textstyle\bigvee_{i=1}^{k} RC_{Ji} \wedge AC_{Ji} \quad (fromDefinition\ 4) \\ &= \textstyle\bigvee_{i=1}^{k} RC_{Ji} \wedge BC_{Ji} \end{aligned}$$

From Theorem 1, we know $RC_I = \bigvee_{i=1}^{k} RC_{Ji} \wedge BC_{Ji}$, where BC_{Ji} is the branch condition under which Ji will branch to I.

Therefore, $BCC_{S,after} = \bigvee_{i=1}^{k} RC_{Ji} \wedge BC_{Ji} = RC_I = BCC_{S,before}$.

Case 2: I is not the only anchor point of S. Assuming S has k anchor points, $I1$, $I2$, ..., Ii, ..., Ik, where $Ii = I$, from Definition 4 we have

$$BCC_{S,before} = (RC_{I1} \wedge AC_{I1}) \vee ... \vee (RC_{Ii} \wedge AC_{Ii}) \vee ... \vee (RC_{Ik} \wedge AC_{Ik}) \qquad (1)$$

After the algorithm is applied, assuming m new anchor points, $J1$, $J2$, ..., Jm, are calculated in place of Ii, we have

$$BCC_{S,after} = (RC_{I1} \wedge AC_{I1}) \vee ... \vee (RC_{J1} \wedge AC_{J1}) \vee ... \vee (RC_{Jm} \wedge AC_{Jm}) \vee ... \vee (RC_{Ik} \wedge AC_{Ik}) \quad (2)$$

Assume there is a source statement S' of which I is the only anchor point. From the proof of case 1 presented above, we have already proven that $BCC_{S',before} = RC_{Ii} \wedge AC_{Ii} = BCC_{S',after} = \bigvee_{i=1}^{m} RC_{Ji} \wedge AC_{Ji}$

Thus, replacing $RC_{Ii} \wedge AC_{Ii}$ with $\bigvee_{i=1}^{m} RC_{Ji} \wedge AC_{Ji}$ in Equation 1, we have $BCC_{S,before} = BCC_{S,after}$.

2.2 Interception points and finish points

When the user sets a breakpoint, the debugger needs to first identify the *interception points* and *finish points* corresponding to the source breakpoint so that it knows where the normal execution should be suspended and where the forward recovery should stop. Note that reaching an interception point of a source breakpoint does not necessarily mean the breakpoint should be reported to the user. Only when an anchor point of the breakpoint is reached during forward recovery can the debugger report the breakpoint.

To calculate the interception points and finish points, information about the original execution order of instructions has to be constructed and preserved during compilation. In the following subsections, an instruction execution-order tracking method is proposed, and the algorithms to calculate the interception points and finish points are described.

2.2.1 Execution order

We propose an instruction execution order tracking method which determines the original execution order of all the instructions and maintains this information during compilation. In our scheme, we do not distinguish execution order between instructions originating from the same source statement. The reason for this is because we are focusing on source-level breakpoints which can only be set at statement boundaries.

To determine the execution order of instructions, one would intuitively think about using source line numbers and column numbers, and annotating each instruction with this information. Although the line number and column number information can determine the execution order of the instructions in the same basic block, it is not sufficient to track the execution order of the instructions across basic blocks. We need to incorporate control flow information to the execution order information. In our scheme, each basic block is assigned an integer number, called *sequence* *number*, which reflects the dynamic execution flow. A basic block with a smaller sequence number will not be executed after another basic block with a larger sequence number without traversing back edges. The derivation and correctness proof of our execution order information is presented in Appendix A.

2.2.2 Interception points

When the user sets a breakpoint at source statement S, all the instructions in the function can be divided into two groups with regard to S based on the execution order information preserved:[2]

pre-breakpoint instructions the instructions which should be executed before S.

post-breakpoint instructions the instructions which should be executed after S, including instructions originating from S.

For a breakpoint set at source statement S, the debugger needs to identify interception points on paths which can lead to the anchor points of S. Assuming all the loops in the optimized code are *monotonic*,[3] for an anchor point I of statement S, the paths that the debugger needs to consider include:

- paths from the function entry point to I without traversing back edges, and
- paths starting from loop headers to I without traversing back edges if I is inside a loop (or loops).

For each path mentioned above, moving forward along the path from its starting point, the first post-breakpoint instruction encountered is an interception point of S.

Referring back to Figure 2 (b), assuming $I3$ is the only anchor point of $S3$, there are two paths leading to $I3$ which the debugger needs to consider: path $P1 = < A, B, C(I3) >$ and path $P2 =$

[2]Instructions which can neither reach S nor be reached from S are also classified as either pre-breakpoint or post-breakpoint based on their execution order information. These instructions are irrelevant as far as the reconstruction of expected variable values at the breakpoint is concerned.

[3]A loop in the optimized code is called *monotonic* if all the instructions in iteration $i+1$ of the loop are supposed to be executed after any instruction in iteration i in terms of the original program execution order, as opposed to *non-monotonic loops* such as modulo scheduled loops [6, 7] where instructions from different iterations of the original loop are mixed together in the same iteration of the new loop. In this paper we only base the discussion of our approach on the assumption that all the loops in the optimized code are *monotonic loops* (such as unrolled loops).

$< B, C(I3) >$. If $I5'$ is the earliest post-breakpoint instruction along $P1$, $I5'$ is an interception point of $S3$. Also, if $I4'$ is the earliest post-breakpoint instruction along $P2$, $I4'$ is another interception point of $S3$.

An algorithm using data-flow analysis to systematically calculate all the interception points with regard to an anchor point is presented in the following.

In the control flow graph G of the function, suppose an anchor point I of statement S is in basic block D and the function entry block is E. To find out the interception points of S with regard to I, we need to first split D into two basic blocks $D1$ and $D2$, where

1. $D1$ is the top portion of D including instructions from the first instruction of D up to the one at I.
2. $D2$ contains the bottom portion of D including instructions from the one immediately following I to the last one.
3. All the D's predecessors become $D1$'s predecessors.
4. All the D's successors become $D2$'s successors.
5. There is no edge directly from $D1$ to $D2$.

Let V be the set of basic blocks which are on the paths leading to $D1$ in graph G (including $D1$).[4] For each basic block B in V, let us define

$gen[B]$ = A one-element set containing the first post-breakpoint instruction in basic block B, if there is any. An empty set, otherwise.

$$kill[B] = \begin{cases} out[B] & \text{if } gen[B] \neq \phi \\ \phi & \text{otherwise} \end{cases}$$

The data-flow equations for in and out sets of B are:

$$\begin{aligned} out[B] &= \bigcup_{S\ is\ a\ successor\ of\ B} in[S] \\ in[B] &= gen[B] \cup (out[B] - kill[B]) \end{aligned}$$

We can use the iterative algorithm shown in Figure 5 to find out the $in[B]$ for each basic block B in V. The union of $in[E]$ and $in[D2]$ is the set of all the interception points of S with regard to I.

[4] V can be obtained through the backward depth-first search from $D1$

```
for each block B in graph G do
    if B is in V then
        in[B] = gen[B];
    else
        in[B] = φ;
    endif
end for
while changes to any of the in sets occur do
    for each block B in V do
        out[B] = ∪_{S is a successor of B} in[S];
        in[B] = gen[B] ∪ (out[B] − kill[B]);
    end for
end while
```

Figure 5: An iterative algorithm for interception point determination

2.2.3 Finish points

To identify finish points, we need to first address an issue about function calls. Due to the surrounding pre-breakpoint instructions, if a post-breakpoint function call which performs some I/O operations such as printing messages to the display screen is executed by the debugger, the user can be confused because the breakpoint was supposed to be set before the function call. Therefore, we do not allow the debugger to execute those post-breakpoint function calls while it does forward recovery. Instead, we will set the finish point before any of function call instructions. This way we might reduce the ability of the debugger to recover the values of some variables because the debugger does not always execute all the pre-breakpoint instructions, but at least it does not confuse the user. Since most compilers have very limited abilities to move code across I/O function calls, this will not be a serious practical issue.

Suppose I is an anchor point of a source statement S. For each different path from I to the function exit point without traversing back edges, the finish point of S on this path is either the earliest post-breakpoint function call or the instruction immediately following the last pre-breakpoint instruction, depending on which one is encountered first. An algorithm using data-flow analysis to calculate all the finish points with regard to an anchor point is designed. The algorithm is similar to the one shown in Section 2.2.2 and is presented in Appendix B.

2.3 Escape points

In our breakpoint implementation scheme, the debugger takes over control at an interception point of a source breakpoint. If an anchor point of the source statement is reached during forward recovery and the anchoring condition is true, the debugger confirms the breakpoint and transfers the control to the user at the finish point. However, before the debugger can make sure that no anchor point of the statement is going to be encountered, it will have to execute instructions (in forward recovery mode) all the way to the end of a function. In order to minimize the number of the instructions executed under forward recovery (which is a lot more inefficient than normal execution), additional information needs to be provided to the debugger so that it can resume normal execution as soon as possible if the breakpoint should not take effect. Another set of object locations (referred to as *escape points*) which are derived from anchor point information is proposed. An *escape point* of a source breakpoint is an object location such that when it is reached during forward recovery, its corresponding breakpoint should not be allowed to take effect and the normal execution is resumed.

For a source statement S, there are two sets of escape points corresponding to it. The first set includes those instructions which can be reached from any of S's interception point(s) but does not lead to any of S's anchor point(s). It is calculated in the following way:

For each anchor point, I, of a source statement S,

step 1 For each interception point of S, find out every different path from the interception point to I without traversing back edges.

step 2 For each path $P = < B_1, B_2, B_3, ..., B_i >$ found in step 1 (where B_1 is the basic block containing the interception point and B_i is the basic block containing I), find out the immediate successors of nodes B_1, B_2, ..., B_{i-1} which are not on any path leading to any anchor point of S without traversing back edges.

step 3 The first instruction of each of the basic blocks found in step 2 becomes an escape point of S.

Figure 6 shows a control flow graph of an example program in which $I2$ is an anchor point of a source breakpoint and $I1$ is the only interception point. We can see that there is only one path from $I1$ to $I2$, which is $< A, B, D >$. We find that basic blocks C and E are the immediate

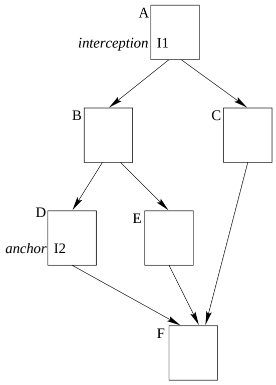


Figure 6: A control flow graph example.

successors of A and B, and they are not on any path leading to $I2$. Therefore the first instructions of C and E are the escape points corresponding to anchor point $I2$.

The second set of the escape points is derived from the anchor points with anchoring conditions other than boolean constant 1. It is calculated in the following way:

For each anchor point I of S which has an anchoring condition other than boolean constant 1 (I must be a conditional branch instruction),

1. If I's anchoring condition is the same as its branch condition, the fall-through target instruction of I which is not on any path leading to any of S's anchor points without traversing back edges is an escape point of S.

2. If I's anchoring condition is the complement of its branch condition, the taken target instruction of I which is not on any path leading to any of S's anchor points without traversing back edges is an escape point of S.

3 Forward recovery model

To recover the expected program behavior, a naive and intuitive forward recovery model is to execute only pre-breakpoint instructions and skip the post-breakpoint instructions. Refer back to the simple example in Figure 1. If this recovery model is used, after taking over control at $I3$ (the interception point), the debugger skips instruction $I3$, $I4$, and $I5$, and executes only instruction $I6$ before it hands over control to the user. Once the user resumes execution, the debugger will go back to execute those skipped instructions.

There are several problems with this model. The first problem is that the instructions in the optimized code cannot simply be re-ordered back to their original execution order because there might be dependency introduced when register allocation is performed after instruction scheduling. Although this problem can be solved by saving the old contents of the destinations of the pre-breakpoint instructions executed, there is another more serious problem. When some pre-breakpoint instructions are moved below a branch instruction which itself is a post-breakpoint instruction, since the debugger will only execute pre-breakpoint instructions under this model, the debugger cannot decide which way to continue the forward recovery without executing the branch instruction.

To avoid these problems, we propose a forward recovery model under which every instruction between the interception point and the finish point is executed and the values updated prematurely are saved. In this model, two important data structures need to be maintained by the debugger during the forward recovery. The first one is the *data history buffer* which keeps track of all the old contents of the destinations of the post-breakpoint instructions executed between the interception point and the finish point. We need these old values to recover the expected values of user variables. An entry in the data history buffer comprises a destination location, which may be a register number or a memory address, and one or more value information records. A value information record of a destination includes the address of the corresponding instruction and the old content of the destination.

The other data structure is called *instruction history buffer* which contains the addresses of the instructions executed between the interception point and the finish point in their dynamic execution order. Each address in the buffer might be annotated with some other information.

The instructions of the debugged program will be executed in either *normal mode* or *forward recovery mode*. The program starts running in normal mode. When an interception point of a source breakpoint is reached during normal execution, the debugger takes over control and the forward recovery mode is entered. From this point on, each instruction will be executed one by one under the debugger's supervision until one of the finish points or escape points is reached.

In the forward recovery mode, before an instruction I is executed, the current content of I's destination along with the address of I will be saved in the data history buffer if I is a post-breakpoint instruction. The address of I is also saved in the instruction history buffer (regardless

```
                address                                     execution order
                2004   I1: r1 = r2 * 2                      1
interception -> 2008   I2: r4 = r3 - r1                     5
                2012   I3: r2 = ld x                        6
                2016   I4: f6 = f2 / 3.0                    2
anchor       -> 2020   I5: y  = st r5                       4
                2024   I6: r4 = r2 - 7                      6
                2028   I7: r2 = 9                           8
                2032   I8: r7 = r3 + 1                      3
finish       -> 2036   I9: ...                              9
```

(a)

Instruction History Buffer

address	annotation
2008	
2012	
2016	
2020	
2024	
2028	
2032	

(b)

Data History Buffer

destination	old value	instr. addr
r4	8	2008
	6	2024
r2	-2	2012
	7	2028
M(y)	0	2020

(c)

Figure 7: (a) Optimized code example (b) Instruction history buffer (c) Data history buffer (the old values in the data history buffer are given arbitrarily in the example).

of whether I is pre-breakpoint or post-breakpoint). If I happens to be an interception point of other breakpoint(s), I's entry in the instruction history buffer will be annotated with this information.

Figure 7(a) shows an optimized program example. For simplicity, we use a single number as the execution order information in this example. Assuming the anchor point of a source breakpoint set by the user is at instruction $I5$. The interception point and the finish point will be set at $I2$ and $I9$ respectively. Once the debugger takes over control at $I2$, each instruction is executed in forward recovery mode until $I9$ is reached. Figure 7(b) and (c) show the resulting instruction history buffer and the data history buffer.

Because some instructions such as load and floating-point operations might cause exceptions during execution, handling the exceptions so that they behave in the way the user expects is very important. If an exception is caused by a post-breakpoint instruction and is posted immediately, the users might be confused because the exception should have occurred *after* the breakpoint. In order not to confuse the user, the debugger should suppress the exceptions caused by post-breakpoint instructions while executing them, and post the exceptions to the user later on. One way to achieve this is for the debugger to provide its own exception handling routine. When an exception occurs in the forward recovery mode, the handling routine provided by the debugger takes over. If the exception is caused by a post-breakpoint instruction, it will be suppressed and the debugger will annotate the entry of the instruction in the instruction history buffer with the exception information so that the exception can be signaled later on. Referring back to the example in Figure 7, if an exception occurs at instruction $I3$, since it is a post-breakpoint instruction, the exception will be suppressed and the entry corresponding to $I3$ in instruction history buffer will be annotated as shown in Figure 8.

Instruction History Buffer

address	annotation
2008	
2012	exception
2016	
2020	
2024	
2028	
2032	

Figure 8: Instruction history buffer.

When a finish point is reached, the debugger stops to answer the user's requests. With the information preserved, our approach can work with a data location tracking method such as the ones proposed in [4] and [5] to provide the expected variable values.

Once the user resumes execution, the debugger will go through the instruction history buffer to check if there is any annotated information and will update both the instruction history buffer and data history buffer. Until an instruction denoted as an interception point of another breakpoint (or other breakpoints) is encountered or the whole instruction history buffer has been processed, the debugger will visit each instruction I in the buffer with the following actions : (1) If the instruction is annotated with exception information, the debugger will signal the exception. (2) The value information record of this instruction's destination (if there is any) is removed from the data history buffer. (3) The entry of this instruction in the instruction history buffer is removed.

If the debugger has visited every instruction in the instruction history buffer without running into another interception point, the normal execution will resume from the finish point.

If an instruction visited is an interception point of another breakpoint, the debugger will continue going through the instruction history buffer in the following way:

1. If the instruction is annotated with exception information, the debugger will first determine the new type (pre-breakpoint or post-breakpoint) of the instruction with regard to the new breakpoint because a post-breakpoint instruction for the old breakpoint might become a pre-breakpoint instruction for the new breakpoint. If the instruction becomes pre-breakpoint, the debugger signals the exception and removes the annotation. Otherwise, the exception remains suppressed.

2. If the type of the instruction is changed (from post-breakpoint to pre-breakpoint), the value information record of this instruction's destination is removed from the data history buffer.

Instruction History Buffer

address	annotation
~~2008~~	
2012	interception point
2016	
2020	
2024	
2028	
2032	

(a)

Data History Buffer

destination	old value	instr. addr
r4	~~8~~	~~2008~~
	6	2024
r2	-2	2012
	7	2028
~~M(y)~~	~~0~~	~~2020~~

(b)

Figure 9: (a)Instruction history buffer (b)Data history buffer (the old values in the data history buffer are given arbitrarily in the example).

Since the finish points of the new breakpoint might be different from those of the old breakpoint, after having gone through the instruction history buffer, the debugger might need to execute more instructions in forward recovery mode until a finish point or an escape point is hit.

To show how the visiting process is working, refer to the example in Figure 7 again. If $I3$ is an interception point of another outstanding breakpoint (whose anchor point is at $I6$), its entry in the instruction history buffer will be annotated with this information as shown in Figure 9(a). When the user wants to resume execution from the current breakpoint, the debugger goes through the instruction history buffer. Since the instruction at address 2008 is not annotated with anything, the debugger removes it from the instruction history buffer and deletes its corresponding entry in data history buffer as shown in Figure 9. The debugger visits the next entry in the instruction history buffer and finds out the instruction at address 2012 is an interception point of another breakpoint. The debugger will keep going through the rest of the instruction history buffer without deleting any entry. Since $I5$ (at address 2020) becomes a pre-breakpoint instruction with regard to the new breakpoint, its entry in the data history buffer is deleted.

Similarly, when an escape point is reached in forward recovery mode, which means the breakpoint should not take effect, the debugger will go through the instruction history buffer in the same way described above.

3.1 Proof of correctness

As we mentioned in Section 1, to provide the expected program behavior, it is essential for the debugger to report the source breakpoints in the order consistent with what the user expects. Our breakpoint implementation scheme handles and reports the source breakpoints in the order of their corresponding interception points being reached. To show that the source breakpoint behavior provided by our approach is correct, we need to prove the interception points of source breakpoints are always reached in the order conforming to the original execution order of their corresponding source statements.

Theorem 4 *For any two source statements $S1$ and $S2$ such that $S1$ has a smaller execution order than $S2$ (that is, $S1$ should be executed before $S2$ if execution flow can reach $S2$ from $S1$), on any path which contains interception points of both $S1$ and $S2$, the interception point of $S2$ will never be reached before the interception point of $S1$.*

Proof: We prove the theorem by contradiction.

Suppose there is a path P contains an interception point of $S1$'s, $I1$, and an interception point of $S2$'s, $I2$, such that $I2$ is reached before $I1$ along P. From the definition of interception point, we know $I1$ is the earliest post-breakpoint instruction on path P with regard to $S1$.

Suppose $I2$ originates from statement B. B should be executed after statement $S2$. Since $S2$ should be executed after $S1$, B should be executed after $S1$. Therefore, $I2$ is also a post-breakpoint instruction with regard to $S1$. Because $I2$ is reached before $I1$ on path P, $I2$ is an earlier post-breakpoint instruction than $I1$ on P with regard to $S1$, which contradicts the assumption that $I1$ is the interception point of $S1$ on path P.

4 Related work

To solve the code location mapping problem in debugging optimized code, there have been different source-to-object mapping schemes proposed such as semantic breakpoints [1], syntactic breakpoints [1], and statement labels [2, 4]. Each of these mapping schemes maps a source breakpoint to a different place in the object code to preserve different kind of source code properties, but the problems suffered by the traditional scheme as we mentioned in Section 1 still remain because all of them map a source breakpoint to a single object location.

Zellweger [8] proposed a method which can correctly map a source breakpoint to every object code location corresponding to the breakpoint. However, her method can only handle programs optimized by *function inlining* and *cross jumping (tail merging)* where a source statement may have more than one instances, or a sequence of machine instructions might correspond to two or more statements. She did not address the problem of mapping a source breakpoint to object locations when the instructions are globally re-ordered or deleted.

In his thesis [2], Adl-Tabatabai proposed to use branch conditions to help the debugger to confirm a source breakpoint when there is no single object location for the debugger to map the breakpoint to. This idea is similar to our anchoring condition scheme. However, he did not provide any in-depth discussion on this topic, nor did he provide algorithms to keep track of the required information during compilation.

Hennessy [9] and Adl-Tabatabai et al. [3] proposed techniques to recover the expected values of variables. Their approaches are similar in concept. They recover the value of a variable by reconstructing and interpreting the original assignment of the variable. The expected value of the variable can be recovered successfully as long as the source operands of the assignment are still available at the object breakpoint. Since both of their approaches adopt a traditional source-to-object mapping scheme, the debugger does not always suspend the execution early enough to preserve the values of the original source operands. Therefore the success rates to recover expected variable values of their approaches are low. Also Adl-Tabatabai's approach can only handle locally optimized code. Hennessy can handle some global optimizations, but only with very limited capability.

There are other research works using different strategies to provide expected program behavior. Gupta[10] proposed an approach to debug code reorganized by a trace scheduling compiler. In this approach the user has to specify the commands for monitoring values before compilation and these commands will be added and compiled into the program. During run time the debugger stops when a monitor command is executed and reports the monitored information to the user. The major problem with this invasive approach is that adding extra code to the debugged program might change the program behavior and consequently mask existing bugs or introduce new bugs.

Holzle, Chambers and Ungar [11] proposed an approach in their SELF programming environment [12] to debug globally optimized code. By dynamically *deoptimizing* code on demand, their debugger can provide full expected behavior. In their approach, the debugger can be invoked only at pre-defined *interrupt points* where the program state is guaranteed to be consistent with what the original program would have. This constraint implies that the optimization can only be performed so that its effects either do not reach an interrupt point or can be undone at that point. Once the debugger is invoked, the function containing the interrupt point is deoptimized so that the debugging requests can be carried out. With the function deoptimized, the program can be stopped at any source point within the function and almost all the typical debugging operations can be supported. The major problem with their deoptimization approach is that the bugs exposed or introduced by the optimizer will be masked.

5 Conclusions

Compiler optimizations cause traditional breakpoint implementation schemes difficulties in two ways: (1) the mapping between source code and object code becomes very complicated, and (2) only the program state of a single point is available, which makes reporting the expected values of user variables very difficult when the program is heavily optimized. In this paper, a new breakpoint implementation scheme for debugging globally optimized code is described. The approach is aimed at solving the problems suffered by traditional breakpoint implementation schemes so that the expected program behavior can be recovered.

In our approach, the debugger takes over the control of execution early to make sure the information required for recovery will not be destroyed permanently. It then moves forward executing instructions under our forward recovery scheme which maintains some data structures to keep track of the program states changed during the forward recovery. Once the debugger has executed all the instructions required, it stops to answer the user's requests. Our scheme can work with a data location tracking method to provide the expected values of variables at the breakpoints. The source breakpoints will be reported to the user in the order specified by the original source program. The behavior of exceptions also meets what the user expects.

A new code location mapping scheme is proposed. The new mapping scheme helps the debugger to determine where to suspend and resume the normal execution and decide if a source breakpoint should be reported. The algorithms and theoretical foundations for constructing and calculating different mappings are presented. A new instruction execution order tracking method at compile time is also described in this paper.

Our breakpoint implementation scheme is primarily targeted at code optimized by techniques involving global code re-ordering, deletion, and duplication. Code optimized by loop transformation techniques such as loop interchange, loop fusion, loop skewing, etc. is not covered by our approach. The feasibility of extending the approach to handle predicated code and advanced ILP optimizations such as modulo scheduling is being explored.

References

[1] P. T. Zellweger, *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, 1984.

[2] A. Adl-Tabatabai, *Source-Level Debugging of Globally Optimized Code*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1996.

[3] A. Adl-Tabatabai and T. Gross, "Detection and recovery of endangered variables caused by instruction scheduling," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 13–25, June 1993.

[4] D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A practical approach to source-level debugging of globally optimized code," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125–134, June 1988.

[5] A. Adl-Tabatabai and T. Gross, "Evicted variables and the interaction of global register allocation and symbolic debugging," in *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 371–383, January 1993.

[6] D. M. Lavery and W. W. Hwu, "Unrolling-based optimizations for modulo scheduling," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 327–337, November 1995.

[7] D. M. Lavery and W. W. Hwu, "Modulo scheduling of loops in control-intensive non-numeric programs," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 126–141, December 1996.

[8] P. T. Zellweger, "An interactive high-level debugger for control-flow optimized programs," *SIGPLAN Notices*, vol. 18, pp. 159–171, August 1983.

[9] J. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 323–344, July 1982.

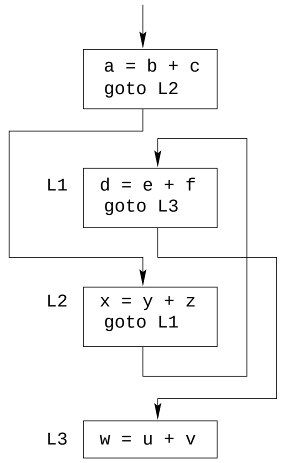
[10] R. Gupta, "Debugging code reorganized by a trace scheduling compiler," *Structured Programming*, vol. 11, pp. 141–150, July 1990.

[11] U. Holzle, C. Chambers, and D. Ungar, "Debugging optimized code with dynamic deoptimization," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 32–43, June 1992.

[12] D. Ungar and R. B. Smith, "SELF: The power of simplicity," in *OOPSLA '87 Conference Proceedings*, pp. 227–241, October 1987.

```
                   line #
    a = b + c        1
    goto L2          2
L1: d = e + f        3
    goto L3          4
L2: x = y + z        5
    goto L1          6
L3: w = u + v        7
```

(a)



(b)

Figure 10: Example program (a) Source program with line numbers (b) Control flow graph

A Derivation and correctness proof of the execution order information

First we want to show why line number and column number information is not sufficient to track the execution order of the instructions across basic blocks. As we can see from Figure 10, although statement L1 has a smaller line number (line 3) than statement L2 (line 5), L2 will always be executed before L1 in the dynamic execution flow as shown in Figure 10(b). Therefore we need to incorporate the sequence number to our execution order information.

To obtain the sequence number, the compiler first builds a control flow graph of the original program. For example, Figure 11(a) shows a control flow graph example. All the back edges in the graph are removed to make the graph acyclic. The compiler then assigns a non-descending sequence number to each one of this partially ordered set of nodes as shown in Figure 11(b) via a topological sort of the graph.[5] Although the sequence number assignment might not be unique, there is only one relative execution order between two basic blocks where the execution control can reach one from the other.

Theorem 5 *In a reducible acyclic control flow graph, there is always a well-defined execution order*

[5]There are some programs whose control flow graphs are irreducible and it's hard to determine the back edges. In this case, we use tail duplication (node splitting) technique to convert an irreducible graph to a reducible graph before we apply our sequence number assigning algorithm.

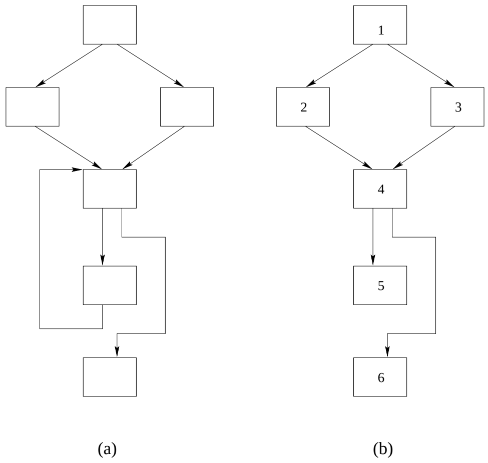


Figure 11: (a) Control flow graph (b) Acyclic control flow graph with sequence numbers assigned to basic blocks

between two basic blocks where execution control can reach one from the other.

Proof: We prove the theorem by contradiction.

Suppose there is a basic block A which can be reached both before and after another basic block B in the control flow graph. There must be a path from A to B and back to A, which makes the graph cyclic and contradicts our assumption. Thus, there is only one execution order between A and B.

Having the sequence number, line number, and column number information associated with each instruction, a simple comparison of the numbers can determine the execution order of instructions.

Before any optimization is performed, sequence numbers will be assigned, along with the line number and column number information, to each instruction. During an optimization phase, the execution order information associated with each instruction remains the same as long as there is no code duplication or code creation optimization performed.

When code duplication optimization which duplicates basic blocks is performed, maintaining the sequence number information depends on if the duplicated code is introduced to a new context. In optimizations such as loop unrolling, function inlining, and loop peeling, the duplicated basic blocks are introduced to a context different from their original one. Their original sequences numbers may no longer be valid in the new context, so we need to dynamically adjust the sequence numbers of the duplicated code and the affected instructions in the surrounding new context to reflect the new execution order. For optimizations such as tail duplication where the duplicated code remains in

```
     foo(int a, int b)
     {
        int t;
S3:     t = a;       (1, 4, 4)
S4:     a = b + 4;   (1, 5, 4)
S5:     b = t;       (1, 6, 4)
     }

     bar()
     {
        int x, y;
S6:     x = 2;       (1, 12, 4)
S7:     y = 3;       (1, 13, 4)
S8:     foo(x, y);   (1, 14, 4)
S9:     y = x + 1;   (1, 15, 4)
     }
```

(a)

```
     foo(int a, int b)
     {
        int t;
S3:     t = a;       (1, 4, 4)
S4:     a = b + 4;   (1, 5, 4)
S5:     b = t;       (1, 6, 4)
     }

     bar()
     {
        int x, y;
S6:     x = 2;       (1, 12, 4)
S7:     y = 3;       (1, 13, 4)
        {
          int a,b,t;
S1':      a = x;     (2, 1, 4)
S2':      b = y;     (2, 1, 4)
S3':      t = a;     (2, 4, 4)
S4':      a = b + 4;(2, 5, 4)
S5':      b = t;     (2, 6, 4)
        }
S9:     y = x + 1;   (3, 15, 4)
     }
```

(b)

Figure 12: Sequence number adjustment for function inlining (a) original C source code (b) functions after inlining. Each statement is annotated with (sequence #, line #, column #).

the old context, we keep the original sequence number information.

To show how the compiler adjusts the sequence number information, we use a function inlining example. Figure 12(a) shows an example C program with two functions, *foo* and *bar*, where *bar* calls *foo*. Each statement is annotated with the execution order information *(sequence number, line number, column number)*. After inlining, statement $S8$ is replaced with a set of statements duplicated from function *foo* as shown in Figure 12(b). In order to maintain the correct relative execution order among instructions originating from function *foo* and function *bar*, we need to change the sequence numbers of all the new statements coming from *foo* and the sequence numbers of the statements which should be executed after $S8$. In Figure 12(b), we can see the sequence numbers of the duplicated statements are all changed to 2 (their original sequence number plus 1, the original sequence number of the function call) and the sequence number of $S9$ becomes 3.

Sequence number adjustment for loop unrolling and loop peeling can be done in a similar fashion.

For optimizations which involve creating new code such as common subexpression elimination, we treat the newly-inserted instructions as if they are from one of the statements involved in the optimization and assign them the same execution order information as the other instructions of the statement. For example, Figure 13(b) shows an optimized program after common subexpression elimination, where $S3$ is newly created code. We assign the execution order information of $S1$ to

(a)	(b)
	`S3: t = a + b  (1, 3, 4)`
`S1: x = a + b  (1, 3, 4)`	`S1: x = t      (1, 3, 4)`
`.`	`.`
`.`	`.`
`.`	`.`
`S2: y = a + b  (1, 8, 4)`	`S2: y = t      (1, 8, 4)`

Figure 13: Execution order information maintenance (a) original C source code (b) program after common subexpression elimination. Each statement is annotated with (sequence #, line #, column #).

$S3$ because we treat $S3$ as if it originates from $S1$.

B A data-flow algorithm for finding finish points

In the control flow graph G of the function, suppose an anchor point I of statement S is in basic block D and the function exit block is E (we assume there is a unique exit block for each function). To find out the finish points of S with regard to I, we need to first split D into two basic blocks $D1$ and $D2$ in the same manner as we do in Section 2.2.2.

Let V be the set of basic blocks which are on the paths from $D2$ to E (including $D2$ and E).[6] For each basic block B in V, let us define

$gen[B]$ = A one-element set containing the instruction which is either the earliest post-breakpoint function call or the instruction immediately following the last pre-breakpoint instruction (depending on which one is encountered first) in basic block B, if there is any. An empty set, otherwise.

$$kill[B] = \begin{cases} in[B] & \text{if } gen[B] \neq \phi \\ \phi & \text{otherwise} \end{cases}$$

The data-flow equations for in and out sets of B are:

$$\begin{aligned} in[B] &= \bigcup_{P\, is\, a\, predecessor\, of\, B} out[P] \\ out[B] &= gen[B] \cup (in[B] - kill[B]) \end{aligned}$$

We can use the iterative algorithm shown in Figure 14 to find out the $out[B]$ for each basic block B in V. $out[E]$ is the set of all the finish points of S with regard to I.

[6] V can be obtained through a depth-first search from $D2$.

for each block B in graph G **do**
    if B is in V **then**
        $out[B] = gen[B]$;
    else
        $out[B] = \phi$;
    endif
end for
while changes to any of the *out* sets occur **do**
    for each block B in V **do**
        $in[B] = \bigcup_{P\, is\, a\, predecessor\, of\, B} out[P]$;
        $out[B] = gen[B] \cup (in[B] - kill[B])$;
    end for
end while

Figure 14: An iterative algorithm for finish point determination