A Novel Breakpoint Implementation Scheme for Debugging 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-98-01

January 1998

Abstract

Providing a clear and correct source-level debugger for programmers to debug highly optimized code has become a necessity with executable binaries increasingly generated by compilers which incorporate advanced optimization techniques. Implementing source breakpoints is essential for providing such a debugger. In this paper, a novel source breakpoint implementation scheme for debugging optimized code is proposed. The approach is aimed at solving the fundamental problem suffered by the traditional scheme. By taking over the control early and executing instructions under a new 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. In order to facilitate this new breakpoint implementation scheme, additional information needs to be generated and maintained during compilation. The compiler support for our approach is also described.

1 Introduction

To fully utilize wide-issue processors, an increasing number of executable binaries are being generated by compilers with advanced optimization techniques. With this increase, it has become a necessity to provide a clear and correct source-level debugger for programmers to debug highly optimized code.

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. In order to support these debugging activities while debugging optimized code, implementing source breakpoints is essential for providing a correct and clear solution to the problem of debugging optimized code.

Traditionally, the debugger maps a breakpoint set at a source statement to an object code location (called *object breakpoint* [1]) and halt the execution when the object breakpoint is reached. The debugger then uses the program state at this point to answer user's requests. However, compiler optimizations cause difficulties to the debuggers using the 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 being 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.

Even though there have been different object breakpoint mapping schemes proposed such as semantic breakpoints [2], syntactic breakpoints [2], and statement labels [3, 4], each of which maps a source breakpoint to a different place in the object code, the fundamental problem of this traditional scheme remains – only the program state of a single point is available. There-

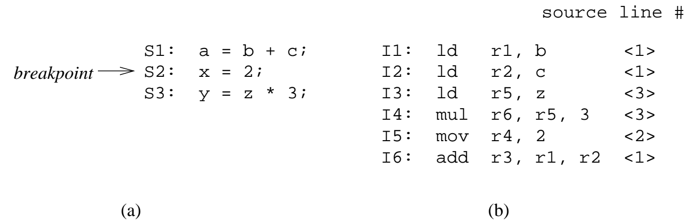

Figure 1: An example program (a) C-style source code (b) Assembly code

fore the optimized code debuggers which 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 [3, 4]. When the values of these variables are requested, the user will be informed that the values are not available at this point. The availability of the variable values decreases when the code is optimized by increasingly aggressive techniques which usually cause more code reorganization.

In this paper, we propose a novel breakpoint implementation scheme which overcomes the problem of 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. It then moves forward in the instruction stream executing 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, it begins to answer user requests. 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 can hand over the control to the user and start taking user's requests. During forward recovery execution, the original contents of `r5`, `r6`, `r4`, and `r3` will be preserved to provide the user with the expected variable values at $S2$.

Although the ultimate goal of our approach is trying to make optimization effects as transparent as possible to the debugger users, it is almost impossible to totally recover the program behavior to what the user expects. Sometimes there is no way to recover the expected value of a variable because the value of the variable does not exist at all after the debugger takes over the control [5]. Sometimes some practical limitations of our approach make us take a conservative approach and therefore the values of the variables can not be recovered. When the expected value of a variable cannot be provided, the debugger will make the users aware of the effects of optimization in the hope of not misleading them.

In order to facilitate our breakpoint implementation scheme, some information must be created and maintained by the compiler and then propagated to the debugger. In this paper, we will also discuss the compiler support for our approach.

In Section 2 we describe the compiler support needed for our approach. We elaborate our breakpoint implementation scheme in Section 3. Section 4 contains our conclusions.

2 Compiler support

2.1 Execution order

Since the debugger in our approach needs to be able to determine which instructions should be executed before a breakpoint and which should be executed after, information about the original execution order of instructions has to be propagated to the debugger. 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. Therefore the execution order of instructions is a partial ordering.

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. As we can see from

Figure 2, 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 2(b). 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.

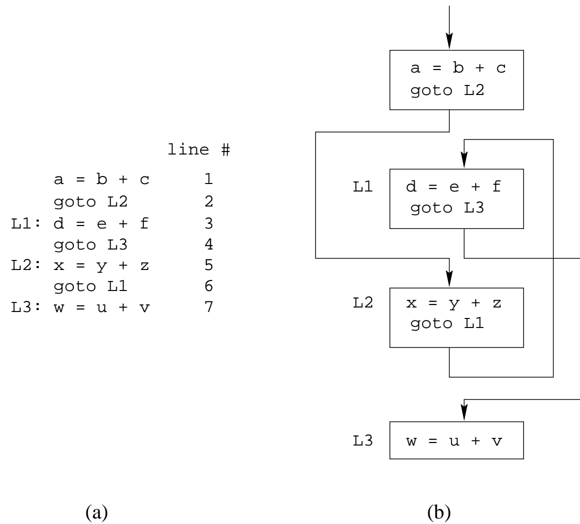

Figure 2: Example program (a) Source program with line numbers (b) Control flow graph

To obtain the sequence number, the compiler first builds a control flow graph of the original program. For example, Figure 3(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 3(b) via a topological sort of the graph.[1] 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 1 *In a reducible acyclic control flow graph, there is always a well-defined execution order between two basic blocks where execution control can reach one from the other.*

proof: Suppose there is a basic block A which can be reached both before and after another basic block B

[1] 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.

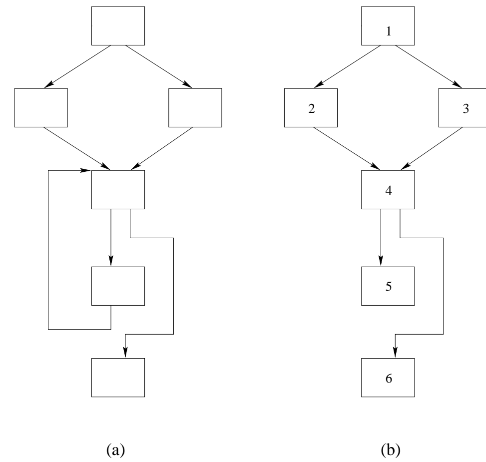

Figure 3: (a) Control flow graph (b) Acyclic control flow graph with sequence numbers assigned to basic blocks

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 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 4(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 4(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 4(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.

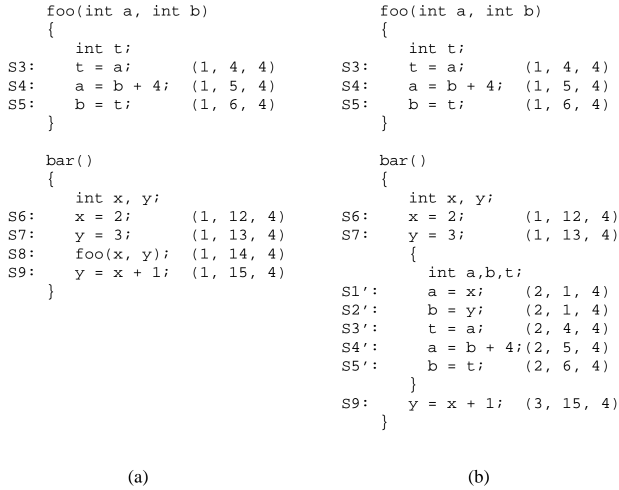

Figure 4: Sequence number adjustment for function inlining (a) original C source code (b) functions after inlining. Each statement is annotated with (sequence #, line #, column #).

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 5(b) shows an optimized program after common subexpression elimination, where $S3$ is newly created code. We assign the execution order information of $S1$ to $S3$ because we treat $S3$ as if it originates from $S1$.

2.2 Anchor point

Since the debugger has to suspend execution early to be able to recover the expected program state, it may do so whether the breakpoint should be allowed to take effect or not. Consider Figure 6. Source statement $S2$ is moved out of loop because variable a is loop-invariant. If a breakpoint is set at source statement $S1$, the debugger needs to suspend execution at $S2'$. By associating *anchor point* information with each source statement

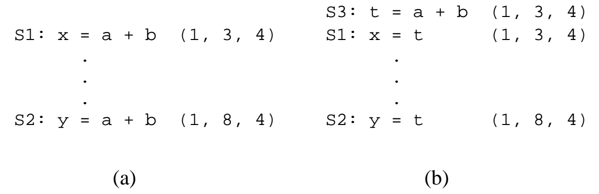

Figure 5: Execution order information maintenance (a) original C source code (b) program after common subexpression elimination. Each statement is annotated with (sequence #, line #, column #).

instance,[2] our approach enables the debugger either to confirm a breakpoint when control flows through the loop body, or to resume normal execution without reporting a breakpoint event to the user when the control flows around the loop.

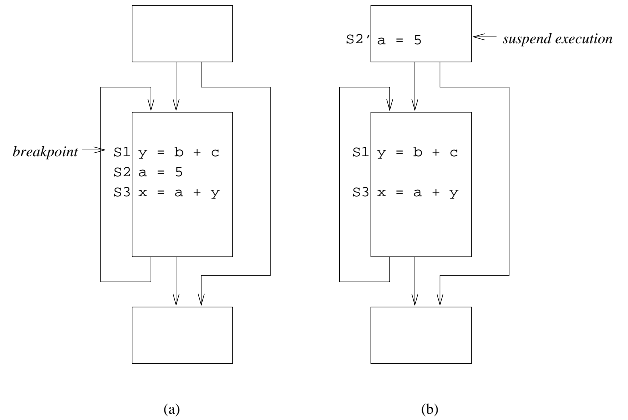

Figure 6: (a) Original program (b) Optimized program after loop-invariant removal

An anchor point of a source statement is a point in the object file. When it is reached during execution, the breakpoint set at that source statement should take effect. During the optimization, the anchor point of a statement instance might be moved, duplicated or might even become conditional. When a *conditional anchor point* of a source statement instance is reached, the breakpoint set at this source statement should take effect only under a certain condition specified by the anchor point information.

Anchor point information for each source statement instance is created and maintained by the compiler. Before discussing how the compiler maintains and updates it, we would like to first define some control equivalent blocks:

Definition 1 *Basic block C is a* Succeeding Equivalent Block (SEB) *of basic block B if C is a successor of B, C post-dominates B, and C is dominated by B.*

[2] A source statement might have more than one instance because of code duplication optimization

```
if B has only one successor C
  and C is a SEB of B
then
  The first instruction of C becomes
  the anchor point of S.
else
  for each of B's predecessors P
     if P is an UPB of B
     then
       the last instruction of P becomes
       an anchor point of S.
     else /* P is a CPB of B */
       the last instruction (which is
       a branch instruction) of P becomes
       a conditional anchor point of S.
       The branch condition under which
       the branch will jump to B is
       recorded.
     endif
endif
```

Figure 7: An algorithm for maintaining the anchor point information of statement S (assuming the basic block being removed is B).

Definition 2 *Basic block C is an* Unconditional Preceding Block (UPB) *of basic block B if B is the only immediate successor of C.*

Definition 3 *Basic block C is a* Conditional Preceding Block (CPB) *of basic block B if C is an immediate predecessor of B but is not an unconditional preceding block of B.*

For example, in the control graph of Figure 8(a), basic block C is a *CPB* of D and E, both basic block D and E are *UPB*s of F, and basic block B is also a *UPB* of D.

Before any code movement or deletion is performed, the anchor point of a source statement instance S is set to the first instruction of S. When the first instruction is moved or deleted, the anchor point is moved to the next instruction in the same basic block. If the instruction being moved or deleted is the last instruction of the basic block, the preceding instruction in the same basic block becomes the anchor point of S. Therefore one instruction may be the anchor point of several statements if the code for an entire statement instance has been moved or deleted. If the instruction being moved or deleted is the only one in the basic block (which means the whole basic block is being removed), we will need to find other control equivalent points to serve as the anchor points of S using the algorithm shown in Figure 7 (assuming the basic block being removed is B).

Note that the algorithm in Figure 7 is based on the following assumption: (1) There is only one entry point of a function. Therefore if B doesn't have a *SEB*, B definitely has a predecessor.[3] (2) A conditional branch cannot be removed. Therefore we don't need to consider the case where a conditional anchor point is removed. If a branch is dead or the condition is a constant, the branch will become unconditional and a conditional anchor point set at this branch instruction will be changed to an unconditional one.

Figure 8(a) shows a control flow graph example where instruction $I3$ is the anchor point of statement instance S and the only instruction in basic block D. If $I3$ is removed due to optimization, the compiler needs to find out the new anchor point(s) of S. Because basic block D has no *SEB*, both instruction $I1$ and $I2$ become the new anchor points of S. Since $I2$ is a conditional anchor point, the compiler needs to record the condition $r1 > 5$.

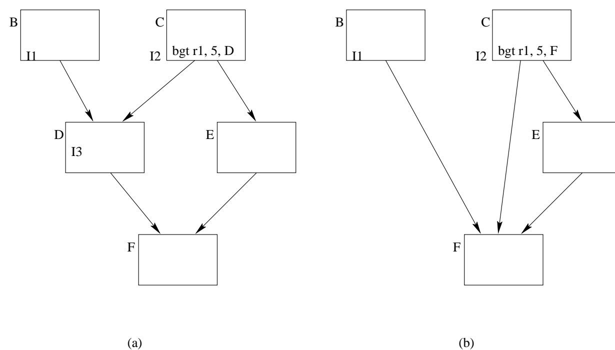

Figure 8: A control flow graph example (a) before removing basic block D (b) after removing basic block D.

The anchor points of source statements are similar to the statement labels mentioned in [3, 4]. We extend the concept of the statement label into the anchor points to indicate whether a breakpoint should be allowed to take effect. We also provide a method to maintain the anchor point information under its extended definition.

3 The breakpoint implementation scheme

3.1 Interception points and finish points

The essence of our breakpoint implementation scheme is that our debugger suspends execution before executing any instruction that should happen after the breakpoint, and then moves forward in the instruction stream executing instructions using a forward recovery technique to ensure that all instructions which should be executed before the breakpoint are complete. Therefore when the user sets a breakpoint, the first action by

[3] If B is an unreachable block, B might not have a *SEB* nor a predecessor. However, the case is irrelevant because breakpoint set inside this block would never take effect anyway.

the debugger is to identify the *interception points* where the normal execution is suspended, and the *finish points* where the debugger begins to take user requests.

When the user sets a breakpoint P at source statement S, all the instructions in the function can be divided into two groups with regard to P:

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

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

The debugger uses the execution order information propagated from the compiler to decide if an instruction is pre-breakpoint or post-breakpoint.

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*, [4] for an anchor point A of statement S, the paths that the debugger needs to consider include:

- paths from the function entry point to A without traversing back edges, and
- paths starting from loop headers to A without traversing back edges if A 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 the interception point on this path.

For example, in Figure 9, assuming the user sets a breakpoint at S and the only anchor point of S is at $I1$, there are two paths leading to $I1$ which the debugger needs to consider: path $P1 = < A, B, C, D(I1) >$ and path $P2 = < C, D(I1) >$. If $I2$ is the earliest post-breakpoint instruction along $P1$, $I2$ is an interception point of the breakpoint. Also, if $I3$ is the earliest post-breakpoint instruction along $P2$, $I3$ is another interception point.

An algorithm using data-flow analysis to find out the interception points with regard to an anchor point is presented in Appendix A.

To identify finish points, we need to first address an issue about function calls. 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 won't be a serious practical issue.

[4]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. In this technical report 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). The approach presented here can be extended to handle *non-monotonic loops* such as modulo scheduled loops [6, 7]. The details about extending our approach will be presented in the full version of the paper.

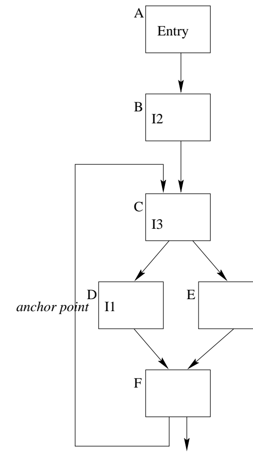

Figure 9: A control flow graph example.

Suppose A is an anchor point of a breakpoint. For each different path from A to the function exit point without traversing back edges, the finish point 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 find out finish points with regard to an anchor point is presented in Appendix B.

3.2 Escape points

Theoretically it is the anchor point which is used to determine if the breakpoint should be allowed to take effect. In practice, the debugger uses *escape points* (which are derived from the anchor point) to do so. When an *escape point* is reached during the execution, its corresponding breakpoint should not be allowed to take effect.

For an anchor point A of a breakpoint, the escape points can be obtained using the following algorithm:

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

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 A), find out the immediate successors of nodes B_1, B_2, ..., B_{i-1} which are not on any path leading to A without traversing back edges.

3. The first instruction of each of the basic blocks found in the above step is one of the escape points of A.

Figure 10 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 successors of A and B, and they are not on any path leading to D. Therefore the first instructions of C and E are the escape points of anchor point $I2$.

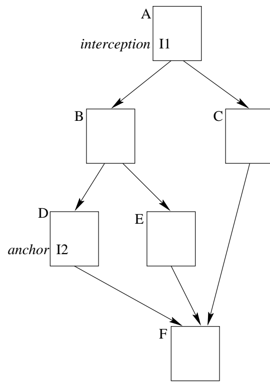

Figure 10: A control flow graph example.

There is another kind of escape points – conditional anchor points. We know that when a conditional anchor point is reached, the breakpoint should take effect only when the condition is true. If the condition is false, the anchor point itself becomes an escape point.

3.3 Forward recovery 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 instructions executed between the interception point and the finish point. We need these old values to be able to recover the expected values of the user variables. An entry in the data history buffer comprises a destination address, 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 instruction which writes to the destination, the old content of the destination, and the instruction type (post-breakpoint or pre-breakpoint).

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 any interception point of a breakpoint set by the user is reached during normal execution, the debugger takes over the control of the execution 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 any one of the finish points or the escape points is reached.

In the forward recovery mode, before an instruction I is executed, the current content of I's destination will be saved in the data history buffer, along with the address and type of I. The address of I is also saved in the instruction history buffer. If I happens to be the interception point of other breakpoint(s), I's entry in the instruction history buffer will be annotated with this information.

Figure 11(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 the control at $I2$, each instruction is executed in forward recovery mode until $I9$ is reached. Figure 11(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

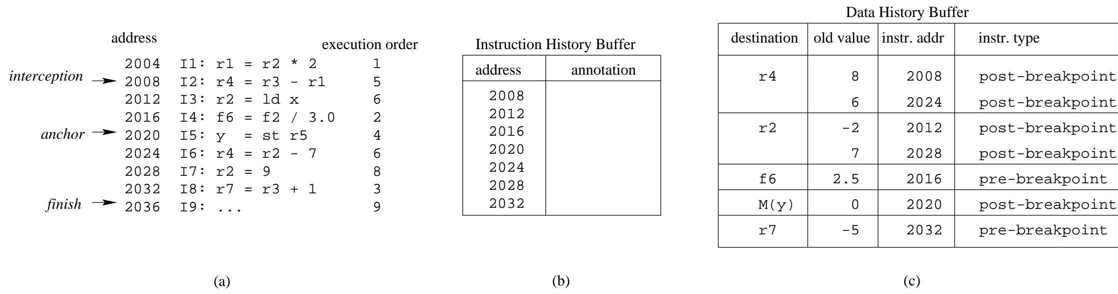

Figure 11: (a) Optimized code example (b) Instruction history buffer (c) Data history buffer (the old values in the data history buffer are given arbitrarily in the example).

on. Referring back to the example in Figure 11, 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 show in Figure 12. For some advanced architectures which support non-trapping versions of instructions [8], the problem can be solved by replacing the post-breakpoint instructions with the non-trapping version of them.

Instruction History Buffer

address	annotation
2008	
2012	exception
2016	
2020	
2024	
2028	
2032	

Figure 12: 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 to provide the expected variable values. Although the data location problem is not the focus of this paper, an introduction to an existing data location tracking method and how our approach works with it are provided in Appendix C.

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 the interception point of another breakpoint (or other breakpoints) is encountered or the whole instruction history buffer has been gone through, 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 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 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 it has become a pre-breakpoint instruction, the debugger signals the exception and removes the annotation. Otherwise, the exception remains suppressed.

2. If the type of the instruction is changed, the type information of the corresponding value information record is changed.

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 11 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 13(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 deletes this entry and its corresponding entry in data history buffer as shown in Figure 13. The debugger visits the next entry in the instruction history buffer and finds out the instruction is an interception point of another breakpoint. The debugger will keep going through the rest of the instruction 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	instr. type
r4	~~8~~	~~2008~~	~~post-breakpoint~~
	6	2024	post-breakpoint
r2	-2	2012	post-breakpoint
	7	2028	post-breakpoint
f6	2.5	2016	pre-breakpoint
M(y)	0	2020	*pre-breakpoint*
r7	-5	2032	pre-breakpoint

(b)

Figure 13: (a)Instruction history buffer (b)Data history buffer (the old values in the data history buffer are given arbitrarily in the example).

without deleting any entry and updating the type of the instructions in the data history buffer. In our example, $I5$ becomes a pre-breakpoint instruction with regard to the new breakpoint.

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.

4 Conclusions

In this paper, a novel breakpoint implementation scheme for debugging optimized code is described. The approach is aimed at solving the fundamental problem of the traditional breakpoint implementation scheme so that the expected values of the user variables can be recovered at source breakpoints. The problem with the traditional scheme is that 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 our approach, the debugger takes over the control of execution early to make sure the information required for the recovery won't 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 the variables.

With our approach, the behavior of the source breakpoints meets what the user expects. Exceptions will be reported in the order that the user expects even though they may be caused out of order internally.

In order to support our novel breakpoint implementation scheme, some information has to be maintained and propagated to the debugger by the compiler. In this paper we also elaborate the compiler support required for our scheme. A new instruction execution order tracking method is proposed and an algorithm for maintaining anchor point information is described.

Due to some practical issues, it is not always feasible to execute all the pre-breakpoint instructions or to suspend the normal execution early enough to preserve all the required information.[5] Sometimes we have to limit our scheme to be conservative in order to make our approach more feasible and efficient. The ability of our scheme to recover the expected values of variables might decrease in some cases because of the conservative approaches. The problem can be alleviated if we can incorporate to our scheme a more advanced data location tracking method which can recover the value of a variable by using the values of other variables.

References

[1] 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.

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

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

[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 interation of global register allocation and symbolic debugging," in *Conference Record of the*

[5]Two practical issues are discussed in detail in Appendix D.

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] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and supercalar processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, October 1992.

A A data-flow algorithm for finding interception points

In the control flow graph G of the function, suppose anchor point A is in basic block D and the function entry block is E. To find out the interception points with regard to A, 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 A.
2. $D2$ contains the bottom portion of D including instructions from the one immediately following A 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 $D1$).[6] 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:

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

We can use the iterative algorithm shown in Figure 14 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 with regard to A.

B A data-flow algorithm for finding finish points

In the control flow graph G of the function, suppose anchor point A 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 with regard to A, we need to first split D into two basic blocks $D1$ and $D2$ in the same manner as we do in Appendix A.

Let V be the set of basic blocks which are on the paths from $D2$ to E (including $D2$ and E).[7] For each basic block B in V, let us define

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

[7] 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
        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 14: An iterative algorithm for interception point determination

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

Figure 15: An iterative algorithm for finish point determination

$gen[B]$ = 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 15 to find out the $out[B]$ for each basic block B in V. $out[E]$ is the set of all the finish points with regard to A.

C Working with a data-location tracking method

In this section we would like to show how our breakpoint implementation scheme can work with a data location tracking method to provide the expected variable values at source breakpoints.

C.1 Data location tracking method

The run-time location of user variables may be altered by optimization. The variable value may be in different places (constant, register, or memory) at different points of execution. Or it may not exist at all. To allow the user to access the value of a variable at breakpoints, the debugger has to know what location holds the value of the variable. Although the data location problem is not the focus of this paper, we use a data location tracking method similar to the approach proposed and implemented by Coutant, Meloy and Ruscetta at HP [4] to show how our breakpoint implementation scheme can recover the expected values of user variables.

In their DOC project, Coutant et al. used a data structure called *range* to track the location information of variables in different ranges of the binary program. For example, in Figure 16(a), variable a lives in three different locations during the execution.

address		source code
2000		a = 3
2004	r8 = 3 + r2	x = a + b
2008	r1 = 3 - r2	a = a - b
2012	r7 = r1 - 6	y = a - 6
2016	r3 = r1 + r8	c = a + x
2020	r2 = r2 + 5	b = b + 5
2024	r5 = r2 + r3	a = b + c
2028	r8 = r7 + r5	x = y + a

(a)

Low Addr	High Addr	Location	Type
2004	2008	3	constant
2012	2016	r1	register
2028	2028	r5	register

(b)

Figure 16: An example code segment and the range structure for variable a

The range structure for variable a is shown in Figure 16(b). By comparing the program counter with the *Low Address* and *High Address* of each range, the debugger can decide where to get the variable value. If the program counter is not in any one of the ranges, which means the variable value is not available at this point, the debugger has to inform the user of this fact.

Range information is calculated based on the live ranges of variables at compile-time. It is considered conservative because the fact that a variable is not live doesn't mean its value is not available. For example, in the program shown in Figure 16, variable a is not live from address 2020 to 2024, but its value is still in register `r1`. Actually Adl-Tabatabai and Gross have proposed a framework using data-flow analysis to provide more accurate data location information[5]. In this paper we still use the range structure to represent the run-time locations of variables but extend the range of a value location to the point where the value is killed. The extended range structure of variable a for the example in Figure 16 is shown in Figure 17.

Low Addr	High Addr	Location	Type
2004	2008	3	constant
2012	2024	r1	register
2028	2028	r5	register

Figure 17: Extended range structure for variable a

C.2 Variable values

We would like to use an example to illustrate how our breakpoint implementation scheme works with the range structure to find out variable values.

Figure 18(a) shows a sequence of *def*s and *use*s of registers along with their corresponding source code. In this example we use a single number as the execution order information. The extended range data structures for variable a and b are shown in Figure 18(b). When the user sets a breakpoint and its anchor point is at $I5$, the debugger takes over the control of the execution at $I4$ and executes the instructions using the forward recovery technique. Once instruction $I9$ is reached, it stops executing to answer the user's requests. The resulting instruction history buffer and data history buffer when $I9$ is reached are shown in Figure 19.

When the user requests for the value of a variable, the debugger will go to the range data structure to find out where the variable value is. It needs to decide what address to use to compare to the address ranges in the range structures. When there is no code re-ordering, the debugger can use the address of the anchor point for every variable. However, when code has been moved around, there doesn't seem to be an address which is right for every variable. In our example, if the user wants to know the value of a, we know address 2032 is a right choice because the assignment which computes the expected value of a is done at address 2028. If the user wants the value of variable b, we have to use the address at or before 2012 because the instruction at 2012 is a post-breakpoint instruction which prematurely updates the value of b.

```
                address                          source code   execution order
                 2000   I1:  def r2                 b =          1
                 2004   I2:  ...                                 2
                 2008   I3:  use r2                   = b        3
interception --> 2012   I4:  def r3                 b =          6
      anchor --> 2016   I5:  ...                                 5
                 2020   I6:  use r3                   = b        7
                 2024   I7:  def r3                 c =          8
                 2028   I8:  def r1                 a =          4
      finish --> 2032   I9:  ...                                 9
                 2036   I10: use r1                   = a       10
```

(a)

variable a

Low Addr	High Addr	Location	Type
2032	2036	r1	register

variable b

Low Addr	High Addr	Location	Type
2004	2012	r2	register
2016	2024	r3	register

(b)

Figure 18: (a)Sample program (b) the extended range structures for variable a and b.

Instruction History Buffer

address	annotation
2012	
2016	
2020	
2024	
2028	

(a)

Data History Buffer

destination	old value	instr. addr	instr. type
r3	2	2012	post-breakpoint
	6	2024	post-breakpoint
r1	5	2028	pre-breakpoint

(b)

Figure 19: (a) Instruction history buffer (b) data history buffer (the old values in the data history buffer are given arbitrarily in the example).

In general, the address used for finding the correct run-time location of variable v can be obtained using the following algorithm:

1. Between the interception point and the finish point, if there is any pre-breakpoint instruction whose destination holds the value of v, find out the latest one of such instructions, say I. We use the address of the instruction following I to find out the run-time location of v. If there is no such instruction, go on to step 2.
2. Between the interception point and the anchor point, if there is any post-breakpoint instruction

whose destination holds the value of v, we use the address of the earliest one of such instructions.

3. If none of the conditions in previous steps exists, we use the address of the anchor point.

After the debugger finds out what location holds the variable value, it will first check if there is an entry of the location in the data history buffer. If there isn't, the debugger goes directly to the location to retrieve the value. If there is, the debugger uses the same address used for comparing range data structure to check if the value is stored in the data history buffer or not. In our example, we know a's value is in register `r1` by looking at the range structure. Even though there is an entry of `r1` in the data history buffer, after doing the address comparison, we know the value of a is not in the history buffer but in the actual register.

D Some practical issues regarding our approach

D.1 Beyond the function scope

We have so far limited the discussion of our approach to within a function. When instructions are moved across a function call, things become a bit complicated. Figure 20(a) shows an example with two functions where function *bar* calls function *foo*. Figure 20(b) shows the program after the optimizer hoists the statement $S2$ above statement $S1$. If a breakpoint is set at statement $S3$ and we only apply our approach within function *foo*, the user won't be able to get the correct value of variable y. Apparently the debugger will have to take over the control of execution as early as at $S2$ in the optimized code in order to solve this problem. To do so, the debugger has to keep track of all the call sites for each function. When the user sets a breakpoint in function *foo*, the debugger will need to go to every caller of *foo* and see if there are instructions moved across the call site. If so, the debugger will have to set the interception points, finish points, and escape points in caller function as if there is a breakpoint set immediately after the call site. If there are multiple layers of functions calls where $F1$ calls $F2$, $F2$ calls $F3$, ..., and Fi calls *foo*, the interception points and finish points should set in the outer-most level of functions where there are instructions moved across the call site.

The problem with this solution is that the instruction history buffer and data history buffer might explode when there is a long chain of function calls. Also the time it takes to run to the breakpoint may be extremely long because lots of the instructions are executed in the forward recovery mode.

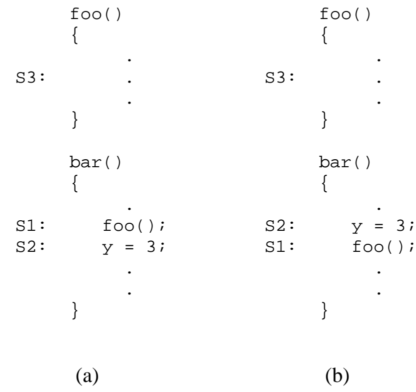

Figure 20: (a)Original program (b)Optimized program.

Therefore, another more practical but less aggressive solution to this problem is to still use the forward recovery technique within a function. As for those variables whose values are updated either too early or too late due to the code movement in the caller functions, we don't try to recover them but just inform the user that the values of those variables are not available at this point because of the optimization. By using this approach, we trade the availability of some variable values for the efficiency and feasibility of the method.

D.2 Loop issues

Another problem occurs when the debugger has to execute all the iterations of a loop in forward recovery mode before it reaches a finish point or an escape point. There are two cases where this problem will happen:

1. There is a loop lying between the interception point and the anchor point. The interception point may or may not be in this loop, but the anchor point is not in the loop. Figure 21(a) shows an example of this case. After taking over the control of execution at $I1$, the debugger has to execute every iteration of the loop in the forward recovery mode before it can reach $I3$.

2. There is a loop lying between the anchor point and the finish point. The anchor point may or may not be in the loop, but the finish point is not in the loop. Figure 22(a) shows an example of this case. After taking over the control of execution at $I1$, the debugger has to execute every iteration of the loop in the forward recovery mode before it can reach $I3$.

Since the number of loop iterations is non-deterministic, the instruction history buffer and data history buffer might potentially explode.

To solve this problem, we can again adopt a less aggressive approach which trades the availability of the

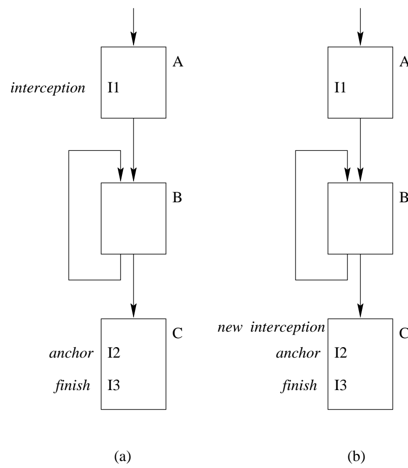

Figure 21: A control flow graph with (a) old interception point (b) new interception point.

values of some variables for the feasibility of our scheme. Our algorithm for finding the interception points and finish points can be modified to avoid these cases. For example, in the case shown in Figure 21(a), the interception point can be set at the beginning of block C as shown in Figure 21(b). Similarly, in the case shown in Figure 22(a), the finish point can be set at the end of block B as shown in Figure 22(a). Some of the variable values might be unavailable at the breakpoint because of this modification.

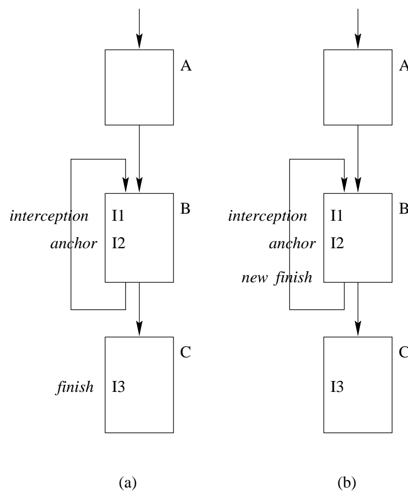

Figure 22: A control flow graph with (a) old finish point (b) new finish point.