Instruction-level Reverse Execution for Debugging

Tankut Akgul and Vincent J. Mooney III
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
{tankut, mooney}@ece.gatech.edu

ABSTRACT

The ability to execute a program in reverse is advantageous for shortening debug time. This paper presents a reverse execution methodology at the assembly instruction-level with low memory and time overheads. The core idea of this approach is to generate a reverse program able to undo, in almost all cases, normal forward execution of an assembly instruction in the program being debugged. The methodology has been implemented on a PowerPC processor in a custom made debugger. Compared to previous work – all of which use a variety of state saving techniques – the experimental results show 2.5X to 400X memory overhead reduction for the tested benchmarks. Furthermore, the results with the same benchmarks show an average of 4.1X to 5.7X time overhead reduction.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms

Keywords

Reverse code generation, reverse execution, debugging

1. INTRODUCTION

As human beings are quite prone to making mistakes, it is very difficult for a programmer to write an error-free program before testing it. For this reason, debugging is an important and inevitable part of software development.

Locating bugs by just looking at the source code is quite difficult. Consequently, a run-time interaction with the program is very useful for debugging. Unfortunately, many of the bugs in programs do not cause errors immediately, but instead show their effects much later in program execution. For this reason, even the most careful programmer equipped with a state of the art debugger might well miss the first occurrence of a bug and might have to restart the program. Furthermore, for difficult to find bugs, this process might have to be repeated multiple times. Even worse, for intermittent bugs due to rare timing behaviors, the bug might not reappear right away when the program is restarted.

Reverse execution provides the programmer with the ability to return to a particular previous state in program execution. By reverse execution, program re-executions can be localized around a bug in a program. When the programmer misses a bug location by over-executing a program, he/she can roll back to a point where the processor state is known to be correct and then re-execute from that point on without having to restart the program. This eliminates the requirement to re-execute unnecessary parts of the program every time a bug location is missed, thus, potentially reducing the overall debugging time significantly.

In this paper, a novel reverse execution methodology in software is proposed. The proposed methodology is unique in the sense that it provides reverse execution at the assembly instruction-level granularity and yet still has reasonable memory and time overheads when the program is being executed. *Note that in the rest of this paper, the word "instruction" refers to an assembly instruction.*

2. BACKGROUND AND MOTIVATION

An execution of a program T on a processor P can be represented by a transition among a series of states S=$(S_0, S_1, \ldots, S_n)$ where a state S_i can be written as a combination of the program counter (PC_i), memory (M_i) and register (R_i) values of P. From this representation, reverse execution of a program can be defined as follows:

Definition 1. Reverse Execution: Reverse execution of a program T running on a processor P can be defined as taking processor P from its current state S_i=(PC_i, M_i, R_i) to a previous state S_j=(PC_j, M_j, R_j), $(0 \leq j < i \leq n)$. The closest achievable distance between S_i and S_j determines the granularity of the reverse execution. If state S_j is allowed to be as early as one instruction before state S_i, then the reverse execution is said to be at the instruction-level granularity. □

The simplest method for obtaining a previous state would be saving that state before it is destroyed. Saving a state during forward execution of a program introduces two overheads: memory and time. Several approaches have been proposed for state saving. In [3, 7], processor states are recorded periodically at certain checkpoints during forward execution of the program. Then, a previous state at a checkpoint can be recovered by restoring that state from the record. However, a previous state at an arbitrary point cannot immediately be recovered, which results in a coarser granularity

PASTE'02, November 18–19, 2002, Charleston, SC, USA.

reverse execution. In incremental state saving [10], on the other hand, instead of recording the whole state, only the modified parts of a state are recorded. However, if modified state space is large, memory and time overheads of incremental state saving might again exceed affordable limits.

In program animation [4, 6], a virtual machine with a reversible set of instructions is constructed. Since these instructions are reversible, the program can be run backwards. However, in program animation, a program can only be interpreted, which slows down the animation considerably, and makes it impossible to execute the program using native machine instructions directly, not even in the forward direction. Moreover, since reversible instructions are usually constructed as stack operations, a significant amount of stack memory may be required in program animation.

Another approach introduced is the source transformation approach [5]. In source transformation, the source code (e.g., in C) is transformed to a reversible source code version excluding destructive statements such as direct assignments. For destructive statements, state saving is applied. Consequently, the execution time and memory requirement of the transformed code are increased. Source transformation does not provide reverse execution at the instruction-level granularity, but instead at the source code (e.g., C) granularity. Moreover, since the original source code is transformed, the program being debugged is no longer the original code, but the transformed code instead. This might be a serious problem in real-time computing where a small change in program code can ruin the timing behavior of the code.

Therefore, our goal is to achieve reverse execution at the native instruction level with low memory and time overheads, which will add a missing feature to state of the art debuggers and will also enable reverse execution of low-level programs such as device drivers.

3. METHODOLOGY

In order to achieve reverse execution of a program at the instruction-level, a new term instruction-level reverse program is introduced as follows:

Definition 2. Instruction-level Reverse Program: Suppose that a processor P attains the series of states $S = (S_0, S_1, \ldots, S_n)$ during its execution of a program T where the distance between two consecutive states is one instruction. S_0 is the initial state before T executes and S_n is the final state just after T quits. Also, suppose that another program T' exists such that when a specific portion of T' is executed in place of T, when P is at a state $S_i \in S$, the state of P can be brought to a previous state $S_j \in S$ $(0 \leq j < i \leq n)$. If T' contains an executable portion for changing the state of P from any state $S_i \in S$ to any other previous state $S_j \in S$ $(0 \leq j < i \leq n)$, then T' is called the instruction-level reverse program of T. □

Taking Definition 2 as a basis, suppose that one wants to generate an instruction-level reverse program T' for a program T. Since the distance between two consecutive states in S is one instruction, T can be represented by an ordered completion of a sequence of assembly instructions $I = (i_1, i_2, \ldots, i_n)$ where $i_k \in I$ $(1 \leq k \leq n)$ changes the state of P from S_{k-1} to S_k . Now, suppose that one could write a set of one or more instructions denoted by I'_k for an instruction $i_k \in I$ such that if I'_k is executed with P being at state S_k, the state of P can be brought back to S_{k-1}. In other words, I'_k can undo the effect of i_k on P's state. We assume that I'_k consists of a set of instructions rather than a single instruction because more than one reverse instruction may need to be generated for reversing the effect of an instruction in I. Then, the effect of the complete sequence I can be taken back by an ordered completion of the sequence of sets $I' = (I'_n, I'_{n-1}, \ldots, I'_1)$ where $I'_k \in I'$ reverses the effect of $i_k \in I$. Therefore, it is sufficient to perform the following three steps to generate a reverse program T' for a program T :

1. Extract the completion order of the instructions in T
2. Generate the sets of reverse instructions of the instructions in T
3. Combine the generated sets of reverse instructions in a way to make those sets complete in the opposite completion order of the instructions in T

Given a program T as an input, the *reverse code generation* (*RCG*) algorithm presented in this paper generates an instruction-level reverse program T' for T by performing the three steps mentioned. Let us call these three steps the RCG steps. Reverse code is generated for every procedure/function in T separately and reverses of procedures/functions are combined by a state saving approach [2]. Therefore, the rest of the paper focuses on the generation of the reverse procedures/functions.

A procedure/function within a program for which a reverse program will be generated is statically analyzed. This static analysis occurs assembly instruction by assembly instruction in the order the instructions are placed by the compiler (lexical order). After an instruction is analyzed, the algorithm goes over the RCG steps consecutively to grow the reverse procedure/function. Some instructions within a loop are analyzed by more than one pass (at most three passes) over the loop body before the reverse code is generated for those instructions.

Note that this paper focuses on assembly level analysis as a first step to provide reverse execution capabilities. However, this research can be extended in a straightforward way to be integrated into a compiler and to connect the reverse assembly execution to reverse source code execution.

In the following subsections, we will explain how the RCG steps are handled in the RCG algorithm. We begin with the explanation of the first RCG step in the next subsection.

3.1 Determining the dynamic control flow information

The static analysis starts with the construction of a *control flow graph* (*CFG*) for each procedure/function in a program T. Each node in a CFG represents a *basic block* (*BB*). A basic block is a single entry, single exit block of a maximal number of consecutive instructions. The important property of a BB is that the instructions within a BB complete in lexical order. Therefore, the CFG construction by itself reveals the completion order of the instructions within the BBs, which partially handles the first RCG step. To satisfy the requirement of the first RCG step for a procedure/function fully, it is necessary to know how control flows dynamically between the BBs of that procedure/function as well.

The easiest way to obtain the dynamic control flow information is via path tracing which is commonly used for variety of dynamic analyses such as dynamic slicing [1]. However, due to memory and time overheads caused by path tracing, we follow another approach which employs the use of control flow predicates. To understand how this is done,

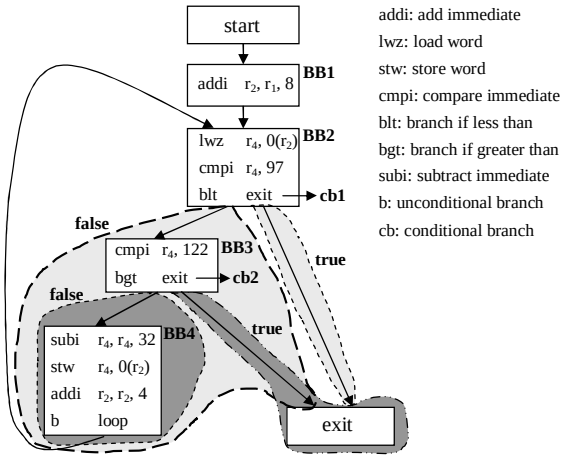


Figure 1: An example control flow graph (CFG).

consider the CFG of a function shown in Figure 1. Let us refer to the function of Figure 1 as F. When the RCG analysis reaches the confluence point of the edges at the entrance of the **exit** block, one needs to know along which incoming edge the **exit** block is dynamically reached, so that the appropriate reverse instructions for F (that will move the control backwards along the actually taken path) can be generated. This is decided by the conditional branch instruction at the end of BB2 which causes the flow of control to be divided into two separate paths before reaching the **exit** block. The predicate expression for this conditional branch instruction is $r_4 < 97$. If the value of the predicate expression $r_4 < 97$ is true, then the **exit** block is reached via one edge (from BB2), otherwise it is reached via the other edge (from BB3). Therefore, knowing the final predicate value (as being true or false) for $r_4 < 97$ attained during the execution of F is sufficient to determine the necessary dynamic control flow information to know, in this case, which edge was traversed to reach the **exit** block.

To obtain the value of a predicate expression attained during forward execution, the RCG algorithm reevaluates the predicate expression during reverse execution. In Figure 1, for example, the value of the predicate expression $r_4 < 97$ can be found once again at the entrance point of the **exit** block by executing the compare instruction "*cmpi* r_4, 97" during reverse execution. Then, one or more conditional branch instructions can be inserted at that point into the reverse code which will take the control backwards according to the reevaluated predicate value. Since the predicate value is obtained during reverse execution, there is no time or memory overhead encountered during forward execution of F. However, in this method, if the value of any variable used in the predicate expression (the value of r_4 in this case) has already been destroyed before reaching the reevaluation point, then that destroyed value must be recovered during reverse execution beforehand. How this recovery is done will be apparent when we explain the second RCG step.

A question that should be answered at this point is how the RCG algorithm determines the predicate expression to use at a point of confluence of edges. For this purpose, the RCG algorithm uses special labels assigned to the edges of the CFG under consideration. As will be explained in Section 3.2, edge labels also assist in finding reaching definitions at a certain program point efficiently. Note that we could also have used a standard *control dependency graph* (*CDG*) [9] analysis to determine the predicate expressions; however, due to the desire to find the predicate expressions and reaching definitions together in an efficient way, edge-labeling is preferred over a CDG analysis. We will now introduce the edge-labeling algorithm and then describe how the predicate expressions are determined.

3.1.1 Edge-labeling algorithm

The RCG algorithm assigns a special label to every forward edge in the CFG of a procedure/function. Backward edges are not considered because giving labels to backward edges helps in the determination of neither the predicate expressions nor the reaching definitions. Since the CFG construction is performed over assembly instructions, a BB in the CFG may have at most two outgoing edges, one for the target path and the other for the fall-through path of a conditional branch instruction ending that BB (i.e., a multiway branch in a high-level programming construct, such as a C "switch" statement, is expressed by a combination of two-way branches at the assembly level).

Each label assigned to an edge indicates the union of one or more closed intervals on a bounded nonnegative integer number axis. An interval $[x,y]$ is named as a *control flow interval* (*CFI*) and is assigned to an edge according to the structure of the program (distinct edges can be assigned the same intervals). As the name CFI implies, each interval specifies (or encodes) a *region* of control flow in the CFG where each *region* of control flow consists of all the BBs and forward edges that reside under *only* one of the branches (true branch or false branch) out of a conditional branch instruction in the CFG. Therefore, each conditional branch instruction (except a conditional branch instruction which is the source of a backward edge) defines two control flow regions (i.e., true region and false region) which are separated from one another by that conditional branch instruction.

Example 1. Control flow regions: In Figure 1, the edge from BB2 to the **exit** block falls into the true region of the conditional branch instruction $cb1$ at the end of BB2. On the other hand, BB3, BB4 and the edges connected to BB3 fall into the false region of $cb1$. As the definition of a control flow region implies, control flow regions can be nested. For instance, in Figure 1, the false region of $cb2$ is nested under the false region of $cb1$; therefore, the false region of $cb1$ constitutes a higher level than the false region of $cb2$. □

By separating the CFG of a procedure/function into a hierarchical structure of control flow regions, the condition under which a specific edge is dynamically visited can be bound to the predicates of the conditional branch instructions that separate those control flow regions.

Figure 2 shows the operations performed on the edges of the BBs in a CFG. We chose to bound the integer number axis between zero and $2^t - 1$ where t is an integer that should be greater than the maximum number of nested conditional branches in a procedure/function body. An unsigned 4-byte integer can represent an integer number axis bounded between zero and $2^{32} - 1$. Therefore, within an unsigned 4-byte integer, a maximum of 31 nested conditional branches can be accommodated, a level of nestedness which is hardly ever seen in a procedure/function. Therefore, for all practical purposes, bounding the integer number axis between zero and $2^{32} - 1$ will be more than enough for the RCG algorithm to function correctly. The code for handling greater than 31 nested conditionals is a special case which will rarely, if ever, be invoked.

(1) $L_{1,1}^{out} = [0,2^t-1]$, *all other* $L_{i,j}^{out} = \phi$
(2) $\forall BB_i \in CFG - \{exit\_BB \wedge start\_BB\}$, *do* {
(3) $\bigcup_{k=1}^{n}[x_k, y_k] = L^{temp} = \bigcup_{j=1}^{|InFwdEdges\ (BB_i)|} L_{i,j}^{in}$
(4) *for* $k = 1$ *to* n {
(5) *if* ($|OutFwdEdges(BB_i)| == 2$) {
(6) $L_{i,1}^{out} = L_{i,1}^{out} \cup [x_k, (x_k + y_k + 1)/2 - 1]$,
(7) $L_{i,2}^{out} = L_{i,2}^{out} \cup [(x_k + y_k + 1)/2, y_k]$
(8) } *else*
(9) $L_{i,1}^{out} = L_{i,1}^{out} \cup [x_k, y_k]$
}
}

Figure 2: Edge-labeling operations.

In Figure 2, the notation $L_{i,j}^{in}$ ($L_{i,j}^{out}$) designates the label of the j^{th} incoming (outgoing) forward edge $\in$ InFwdEdges ($\in$ OutFwdEdges) of the i^{th} basic block. The edge-labeling algorithm starts with the **start** block and traverses all other BBs in the CFG (except the **exit** block) in topological order ignoring any backward edges. Therefore, all predecessors of a node in the CFG are visited before the node itself. Since backward edges are ignored, the tail node of a loop (i.e., the source node of the backward edge) is excluded from the predecessors of the header node of that loop.

First, the algorithm assigns to the outgoing edge of the **start** block the label $[0,2^t - 1]$ which indicates all of the bounded nonnegative integer number axis and initializes the labels of all other forward edges in the CFG to empty sets (line 1 of Figure 2). Then, for any BB in the CFG except the **start** block and the **exit** block, L^{temp} is set to be the union of the labels of the incoming forward edges of that BB where the union operation is performed on the intervals indicated by the labels (line 3 of Figure 2). After the union operation, if the BB has two outgoing forward edges, each interval designated by L^{temp} is divided into two equal portions. Then, the union of the lower portions (coming from each interval) is assigned as a label to the outgoing forward edge in the fall-through path (line 6 of Figure 2). The union of the upper portions, on the other hand, is assigned as a label to the outgoing forward edge in the target path (line 7 of Figure 2). If the BB has one outgoing edge, L_{temp} is assigned to that edge without any change (line 9 of Figure 2).

Example 2. Edge-labeling algorithm: Figure 3 shows the CFG of Figure 1 with its edges labeled. The edge-labeling algorithm starts labeling the edges with the outgoing edge of the **start** block. For this example, the parameter t shown in Figure 2 is chosen as 8. Therefore, the outgoing edge of the **start** BB is given the label [0,255]. Since BB1 has only one outgoing forward edge, [0,255] is assigned to BB1's outgoing forward edge without any change. BB2 has two outgoing forward edges, therefore, [0,255] is divided into two equal portions [0,127] and [128,255] and each portion is assigned to one of the outgoing edges. The same process is repeated for the other BBs as well. All the CFIs formed are shown in Figure 4. Note that in this example, each label consists of a single interval. □

3.1.2 Extracting the control flow predicates

A confluence point P in a CFG is dynamically reached along an incoming edge e if the innermost control flow region in which e resides is dynamically visited. Therefore, the predicate expression which, when true, causes P to be reached via e will simply be an appropriate combination of the predicates of the relevant conditional branch instructions which cause the innermost control flow region of e to be visited. Since the edge labels encode control flow regions, determination of the hierarchy of the control flow regions in which e resides and thus the relevant conditional branch instructions to use can be accomplished very easily by using the edge labels (for more information, please refer to [2]).

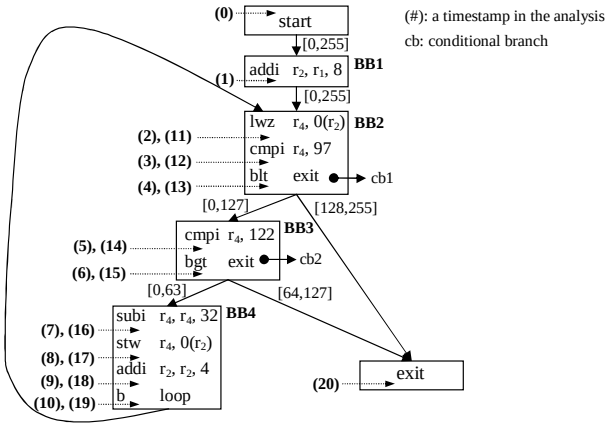


Figure 3: The example CFG with labeled edges.

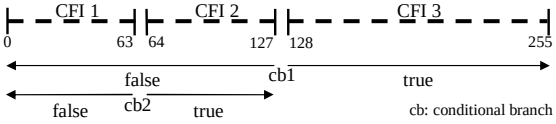


Figure 4: The control flow intervals.

Example 3. Control flow predicate determination: Suppose that we want to find the predicate expression that controls via which incoming edge the **exit** block in Figure 3 will be reached dynamically. We expect this predicate expression will be found to be $r_4 < 97$. The incoming edge labels of the **exit** block are [64,127] and [128,255] for the left and the right incoming edges, respectively. As seen in Figure 4, [64,127] corresponds to the CFI where the predicate, $r_4 < 97$, of the conditional branch *"blt exit"* is false and the predicate, $r_4 > 122$, of the conditional branch *"bgt exit"* is true. On the other hand, within [128,255], only the predicate $r_4 < 97$ is true. Therefore, the **exit** block will dynamically be reached via the left incoming edge if the predicate $r_4 < 97$ is false and the predicate $r_4 > 122$ is true. On the other hand, the **exit** block will dynamically be reached via the right incoming edge if the predicate $r_4 < 97$ is true. Since the CFI which corresponds to the false value of the predicate $r_4 > 122$ is not spanned by any of the incoming edge labels, the value of predicate $r_4 > 122$ is irrelevant in this case. Therefore, the value of the predicate expression $r_4 < 97$ alone distinguishes the dynamically taken edge. □

Note that since backward edges are not labeled, the predicate expression which determines whether a loop header block will dynamically be reached via an incoming backward edge or a forward edge cannot be found by the method explained. Therefore, for loop header blocks only, we insert a loop counter into the original code by which we can differentiate the first iteration of the loop from the rest of the iterations (see the end of Example 6 in Section 3.3).

3.2 Reverse code generation

As stated in the beginning of Section 2, the processor state consists of the program counter, registers and memory. The recovery of the program counter value is already covered by

the first RCG step by which the dynamic control flow information of a procedure/function is obtained. Therefore, the remaining parts of the state to recover are the register and memory values modified by the instructions of the procedure/function under consideration. The second RCG step, which will be explained in this section, involves the generation of a set of one or more reverse instructions for every instruction which modifies a register or a memory location in a procedure/function.

To recover a destroyed value D of a variable (a register or a memory location) V modified by an instruction α in a procedure/function, first of all one needs to know at what point in the procedure/function D might be assigned to V. This is exactly the same problem as finding the statically reaching definitions of V at a point just above the instruction α.

To determine the statically reaching definitions at a program point, the RCG algorithm employs a very efficient method called *value renaming* which refers to giving a different name to every value attained by a register or a memory location. Value renaming is same as the renaming operation in *static single assignment* (*SSA*) form generation [9]. Then, as mentioned before in Section 3.1, the RCG algorithm uses the labels on the edges of the CFG to efficiently find the reaching definitions at each procedure/function point.

After determining the reaching definitions of variable V which was modified by instruction α, the RCG algorithm generates the set of instructions which undo the effect of α. For this purpose, the RCG algorithm uses a *directed acyclic graph* (*DAG*) that is separately built for each procedure/function. This DAG keeps the data dependency information about which variables a definition of V uses and by which variables a definition of V is used. Then, the destroyed value D of V can be recovered by using this data dependency information as will be explained in Section 3.2.3.

Therefore, in the following three subsections, we will describe value renaming, determination of the statically reaching definitions and finally the operations on a DAG. More detailed descriptions of these processes can be found in [2].

3.2.1 Value renaming

In our approach, different renamed values are designated by r_i^j and m_k^j for registers and memory locations, respectively. Here, i indicates a specific register, k indicates a specific memory location, and j ($j = 0, 1, 2, \ldots$) indicates the unique index of a particular renamed value (renamed during program analysis). Index '0' is always used to refer to the initial value of a register or a memory location.

A memory store local to a procedure/function can be expressed as a summation of the fixed value of the frame pointer (or the stack pointer if the frame pointer is not available as a dedicated register) during a procedure/function execution and the offset used for the store. Global stores can be expressed in a similar way, but by using the fixed base address of the data section in place of the frame pointer. Note that if an offset of a store cannot be determined statically, the stored value is given a distinct name, and, to be conservative, that ambiguous store is assumed to be able to modify any other memory location.

Example 4. Value Renaming: Consider the following instruction sequence from the function F shown in Figure 3:

```
addi r2, r1, 8        //r2 = r1 + 8
stw  r4, 0(r2)        //mem[r2 + 0] = r4
addi r2, r2, 4        //r2 = r2 + 4
```

The initial values in registers are given the names r_2^0, r_1^0 and r_4^0. Then, the first instruction generates a new value designated by r_2^1 for register r_2 and the third instruction generates another value designated by r_2^2 for the same register. The second instruction, on the other hand, writes the contents of r_4 into the memory location to which r_2 points. Note that the code in Figure 3 is compiled for a PowerPC target (the PowerPC does not have a dedicated register for the stack pointer) with register r_1 holding the stack pointer. Therefore, the compiler ensures that the value of r_1 is fixed during the execution of F. As seen in Figure 3, r_2 is set to r_1+8 in BB1 before the first iteration of the loop and is incremented by '4' in BB4 at each iteration of the loop. Therefore, the target addresses of the memory stores (stores made by *stw* r_4, $0(r_2)$ in BB4) at each iteration of the loop can be expressed relative to the stack pointer as follows: r_1+8, r_1+12, r_1+16,.... Since these locations are known to be distinct locations due to the fixed value of r_1 in F, the corresponding renamed memory values for the memory stores at different loop iterations will be m_0^1, m_1^1, m_2^1, ... □

3.2.2 Determination of the reaching definitions

Statically reaching definitions at a point in a procedure/function are found by the labels on the forward edges of the CFG of that procedure/function. Therefore, the RCG algorithm labels all the forward edges of the CFG under consideration prior to reaching definition determination.

Since the algorithm determines reaching definitions while analyzing the instructions, loop carried definitions cannot be determined before the whole loop is analyzed, which requires at least one analysis pass over the loop body. Therefore, during the first traversal of the loop, the RCG algorithm generates reverse code by using the definitions that come from outside of the loop only (i.e., reverse code is generated for the first iteration of the loop only) and during the next traversal, the loop carried definitions are used. However, passes over the loop body might not be limited to two due to a loop constraint which will be explained in Section 3.2.4.

To determine the statically reaching definitions, a table called the *renaming table* is kept by the RCG algorithm (see Figure 5 for an example). The renaming table has a record for every physical location (e.g., r_1, r_2, m_1, ...) that has been modified in a procedure up to the instruction currently being analyzed. As more locations are modified, more records are dynamically added to the renaming table. Every record in the renaming table has a field for each CFI produced in a procedure/function body. Initially, all the fields in a newly added record in the renaming table contain the initial value of the corresponding physical location. The field(s) to be used for an entry when analyzing a basic block BB_i is (are) determined by applying the following rule:

$$[x_1, y_1] \cup [x_2, y_2] \cup \ldots [x_n, y_n] = \bigcup_{j=1}^{|InFwdEdges(BB_i)|} L_{i,j}^{in}$$

$$Fields \mapsto \{c | x_k \leq L(c) \wedge U(c) \leq y_k, 1 \leq k \leq n, c \in CFIs\}$$

L(c) and *U(c)* designate, respectively, the lower and upper bounds of a CFI (as stated at the beginning of this section, CFI calculation has been done already by an initial pass over the procedure/function). According to the above rule, a renamed value generated within a BB is written into the renaming table fields that correspond to the CFIs spanned by the labels on all incoming forward edges of that BB.

In addition to the rule above, the RCG algorithm performs three more actions. First, as stated in Section 3.2.1,

we assume that an ambiguous memory store (e.g., using an ambiguous pointer) may change any memory location. Due to this assumption, a renamed value generated for an ambiguous memory store and entered into some renaming table field(s) according to the rule above deletes the entries in the same field(s) of the records belonging to other memory locations. Second, as mentioned in Section 3.1.1, distinct edges can be assigned the same labels. If not properly anticipated, this could result in an incorrect determination of reaching definitions within a BB in some special situations (for an explanation of these special situations, please refer to [2]). To deal with these special situations, the RCG algorithm merges distinct definitions of a variable reaching a confluence point in the CFG under a *pseudo definition*. The pseudo definition is renamed as any other ordinary definition and the renaming table entries that correspond to the combined reaching definitions are overwritten by the value given to the pseudo definition. As will be described in the next subsection, the combined reaching definitions are not thrown away, but are represented in the DAG. Third, since backward edges are not labeled, edge labels cannot be used directly to find the loop carried reaching definitions. Therefore, at the end of each pass over a loop body, the RCG algorithm carries the definitions reaching the end of the loop tail block to the beginning of the loop header block [2].

Finally, the statically reaching definitions at a point P during the analysis can be determined simply by querying the renaming table fields at P. If P is the entrance of a basic block BB_i, the statically reaching definition of a variable V along an incoming forward edge e_j of BB_i is the definition in the renaming table fields corresponding to the CFIs that are spanned by the label on e_j. If P is inside BB_i, on the other hand, statically reaching definition of V is the definition in the renaming table fields which correspond to the CFIs spanned by the labels on all of the incoming forward edges of BB_i (there will be only one statically reaching definition of V along an e_j or within a BB_i because multiple definitions are merged under a pseudo definition at confluence points).

Example 5. Determination of the reaching definitions: Consider the CFG in Figure 3. The renaming table generated for this CFG after two passes over the loop (excluding the first pass over the whole program to generate the CFGs and the CFIs) is shown in Figure 5. The renaming table shows the analysis timestamps adjacent to a renamed value when that value is generated (timestamps are shown in parentheses in Figure 3). The timestamp value increments by one after each instruction in a procedure/function is scanned. For clarity, overwritten entries are also shown in the renaming table (Figure 5). Recall that the RCG algorithm finds the reaching definitions separately for different iterations of a loop. Therefore, assume that the loop is being traversed the very next time after generation of the CFG and the CFIs. Suppose that we want to determine the reaching definitions of register r_4 at the entrance of BB3 at timestamp '4'. For this purpose, the RCG algorithm queries the renaming table fields that correspond to the CFIs spanned by the label [0,127] of the incoming forward edge to BB3. As seen in Figure 4, the label [0,127] spans CFI1 and CFI2, and the renaming table fields in Figure 5 corresponding to these CFIs hold the value r_4^1: the value r_4^1 has been generated by the instruction *"lwz* r_4, $0(r_2)$" at timestamp '2' and has been written into the renaming table fields corresponding to the CFIs spanned by the incoming forward edge label of BB2 – the BB which holds the instruction *"lwz* r_4, $0(r_2)$." The value r_4^1 is indeed the reaching definition of r_4 at the entrance of BB3 at the first iteration of the loop. □

	r_1	r_2	r_4	m_0	m_1
CFI 1	r_1^0 (0)	r_2^0 (0) r_2^1 (1) r_2^2 (9) r_2^3 (18) r_2^4 (19)	r_4^0 (0) r_4^1 (2) r_4^2 (7) r_4^3 (11) r_4^4 (16)	m_0^0 (0) m_0^1 (8) m_0^2 (19)	m_1^0 (0) m_1^1 (17) m_1^2 (19)
CFI 2	r_1^0 (0)	r_2^0 (0) r_2^1 (1) r_2^2 (10) r_2^4 (19)	r_4^0 (0) r_4^1 (2) r_4^2 (10) r_4^3 (11) r_4^4 (19)	m_0^0 (0) m_0^1 (10) m_0^2 (19)	m_1^0 (0) m_1^2 (19)
CFI 3	r_1^0 (0)	r_2^0 (0) r_2^1 (1) r_2^2 (10) r_2^4 (19)	r_4^0 (0) r_4^1 (2) r_4^2 (10) r_4^3 (11) r_4^4 (19)	m_0^0 (0) m_0^1 (10) m_0^2 (19)	m_1^0 (0) m_1^2 (19)

Figure 5: The renaming table for the code example.

3.2.3 Operations on a DAG

In this section, we will describe how a DAG is used to recover a destroyed value at a procedure/function point.

After the RCG algorithm generates a new renamed value (either for an instruction or for a pseudo definition), a new node for that renamed value is added to the DAG, G=(N,E), which is constructed for the procedure/function under consideration. Moreover, to recover a destroyed value, the RCG algorithm should specify the relationship of the destroyed value with the other values generated in the procedure/function. Therefore, the RCG algorithm adds edges to the DAG to connect the nodes that have a data dependency inbetween. N and E include the following:

- N=$\{R,M\}$ where R and M are the sets of renamed register and memory values, respectively.
- There is a directed edge $e_{ij} \in E$ from node $n_i \in N$ to node $n_j \in N$ designated by $n_i \rightarrow n_j$ if (1) n_i and n_j are the renamed values for target and source operands of an instruction α, respectively, or (2) n_i is a renamed memory value and n_j is a renamed register value determining the location of n_i, or (3) n_i and n_j are the renamed values for a pseudo definition and a combined definition under that pseudo definition, respectively.

We also apply some annotations on particular nodes and edges in the DAG to obtain the necessary information for the recovery of a destroyed value: in cases (1) and (2) above, node n_i is annotated with the address of α to show for which instruction n_i is generated. In case (3) above, node n_i is annotated by a special select (S) operator to show that n_i is generated for a pseudo definition. Also, since a pseudo definition cannot be directly used to recover a destroyed value (but one of the combined definitions represented by that pseudo definition can be), in case (3) above, the condition (or the predicate expression) under which the pseudo definition n_i will be equal to the renamed value n_j is attached as an annotation to the edge e_{ij} from node n_i to node n_j.

A node n_i in the DAG can have at most one of the following attributes at a point P: *killed*, *available* and *partially-available*. Node n_i is killed at P if the value of n_i does not reach P; n_i is available at P if the value of n_i reaches P along all paths; and n_i is partially available at P if the value of n_i reaches P along some path controlled by a predicate expression (i.e., n_i is the value of a combined definition).

Suppose that an instruction α_{dest} destroys the value D of a variable V at a procedure/function point. Let us name the point just before and after α_{dest} as P and P', respectively. In order to recover D, the RCG algorithm tries to find the reaching definition of V at point P using the renaming table (remember that there exists only one reaching definition of a variable within a BB due to the merging operation). A definition cannot be found only if the corresponding entry/entries was/were deleted in the renaming table due to

an ambiguous memory store (see Section 3.2.2). In this case, D is recovered by state saving. If a definition can be found, on the other hand, the RCG algorithm finds in the DAG the node that corresponds to the found reaching definition. Suppose that the found node is n_i. Since D is destroyed by α_{dest}, node n_i is killed at point P'. Now, if one or both of the following are true at P', we can recover n_i by generating the appropriate reverse instructions.

(a) All n_j's, where there exists an edge $n_i \rightarrow n_j$, are available and n_i and n_j's are the values of the operands of an instruction α.

(b) An n_j, for which there exists an edge $n_j \rightarrow n_i$, is available and all n_k's, $n_k \neq n_i$, for which there exists an edge $n_j \rightarrow n_k$, are available as well. Moreover, n_i, n_j and all n_k's are the values of the operands of an instruction β which allows n_i to be extracted out of β.

If (a) holds, n_i can be recovered at P' by executing α without any change. On the other hand, if (b) holds, n_i can be recovered at P' by extracting n_i out of β. In addition, if any node n_j that is needed for recovering n_i is partially-available (i.e, n_j is the value of a combined definition), controlled by a predicate expression Υ, then n_i might be partially recovered at P' (the predicate expression Υ is obtained by the annotations on the edges coming to n_j in the DAG). To recover n_i totally, n_i must be partially-recoverable for all values of Υ. In this case, the reverse code for recovering n_i will be gated by Υ. If Υ is destroyed itself, the nodes determining Υ's value must be recovered as well. Finally, note that these actions can be applied recursively, that is, if a node n_j that is required to recover n_i is killed, then n_i might still be recovered by recovering n_j first. If the number of recursions exceeds a predetermined number which is set by the user, or the recovery of a node requires the knowledge of the value of an external input of the procedure/function under consideration, we save state to recover the killed node.

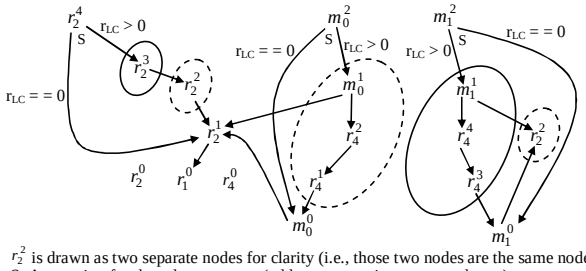
3.2.4 Handling loops

There is a constraint for the generation of the reverse code within loops: A killed node during the analysis of a loop must be recovered only by using (or manipulating) the instructions within the loop. If an external instruction to the loop is used instead, the killed node will be recovered only for a single iteration of the loop. In this case, another pass over the loop body is necessary to construct additional nodes (associated with the internal instructions to the loop) to be used in the recovery of the killed node. The passes over the loop are limited to three to limit the time cost of the RCG algorithm. If a suitable internal instruction cannot be found within three passes, we save state to recover the killed node.

3.3 Putting it all together

The next example illustrates reverse code generation.

Example 6. Reverse code generation: Figure 6 shows the DAG that is constructed after analyzing (with two passes over the loop) all the instructions in the function F of Figure 3. We will again use the timestamps shown in Figure 3 to refer to the analysis instances. As an example, consider the analysis point reached after scanning *"lwz* r_4, $0(r_2)$" at timestamp '2'. The analysis first finds the reaching definition of r_4, r_4^0, by querying the renaming table fields spanned by the incoming edge label [0,255] (the renaming table is shown in Figure 5). Then, the newly generated value of r_4, r_4^1, is entered into the same fields according to the rule described in Section 3.2.2: the result can be seen in all the $r_4^1(2)$ entries in Figure 5. Next, a node



r_2^2 is drawn as two separate nodes for clarity (i.e., those two nodes are the same node).
S: Annotation for the select operator (address annotations are not shown).
Nodes generated within the loop are encircled (dotted: first pass, solid: second pass).

Figure 6: The DAG for the code example.

for r_4^1 is constructed in the DAG and is connected to the node m_0^0 (m_0 designates the memory location at r_1+8). Finally, r_4^0 should be recovered. Since r_4^0 is an input to F and F has no instruction associated with r_4^0, r_4^0 has to be recovered by state saving. Therefore, r_4^0 can be recovered by the load instruction *"lwz* r_4, $mem2$" where $mem2$ is the location where r_4^0 is saved in F. However, since *"lwz* r_4, $mem2$" is not an instruction within the loop, the loop condition mentioned in Section 3.2.4 is violated (i.e., r_4 is recovered only for the first iteration of the loop); therefore, another pass over the loop body is necessary. When the analysis reaches the same point at timestamp '11', the value of r_4 to be recovered is now r_4^2. Fortunately, r_4^2 can be recovered by using internal instructions this time: r_4^2 has an incoming edge from m_0^1. Although, m_0^1 is available, r_2^1, the other node m_0^1 is connected to, is killed. However, condition (b) given in Section 3.2.3 holds for r_2^1 and thus r_2^1 can be recovered into a temporary register r_t by using the available node r_2^2 and with staying in the loop (r_t is used instead of r_2 to preserve the value r_2^2 of r_2). The instruction for recovering r_2^1 will then be *"subi* r_t, r_2, 4" which extracts r_2^1 out of the addition instruction *"addi* r_2, r_2, 4" (*"addi* r_2, r_2, 4" is found by the address annotation on r_2^2). Now, condition (b) holds for r_4^2 as well, and r_4^2 can be recovered for the rest of the iterations of the loop by executing the instruction *"lwz* r_4, $0(r_t)$." A loop counter (r_{LC}) inserted into the original code is used for differentiating between the loop iterations. Note that if no free registers are available to be used as r_{LC} and/or r_t, any occupied registers can be freed up by state saving [2]. □

The final step of the RCG algorithm is to combine the sets of reverse instructions to generate the reverse procedure/function. This process is illustrated in Figure 7. Since instructions within a BB complete in lexical order, for every BB in function F (shown on the left in Figure 7), the RCG algorithm places the generated sets of reverse instructions into F' (shown on the right in Figure 7) in reverse lexical order. Reverse lexical order implies that the sets of reverse instructions within the BBs of F' are placed in bottom-up order (i.e., the first generated reverse set is placed at the very bottom, the second set is placed above the first set and so on). Then, the RCG algorithm combines the reverses of BBs in such a way that control flows between the BBs of F' in the opposite order it flows between the BBs of F. This is done by inverting the edges of the CFG of F. Consequently, a join point of edges in F typically becomes a fork point of edges in F', and vice versa. Note that since the reverse of BB3 in F happens to be empty [2], the inverted versions of the two incoming edges of the **exit** block in F go to the same point in F'. Therefore, these inverted edges are merged together into a single edge. If this were not the case, a conditional branch instruction of which predicate is determined as explained in Section 3.1.2 would be inserted at the end of the **start** block in F'.

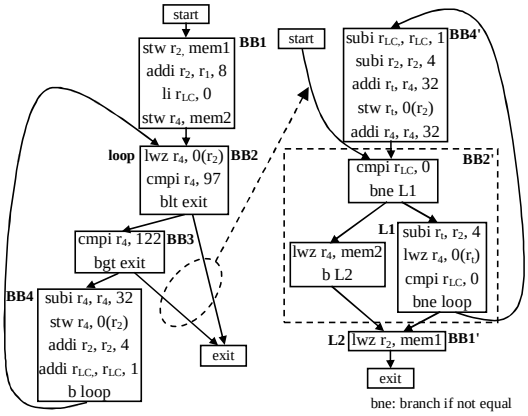


Figure 7: The forward CFG (left) and the reverse CFG (right).

4. EXPERIMENTAL RESULTS

We tested the RCG algorithm on an evaluation board with a PowerPC (MPC860) processor. Also, to test reverse execution on a debugging session, we implemented a low-level debugger tool with a graphical user interface that provides debugging capabilities such as breakpoint insertion, single stepping and register/memory display. The debugger runs on a PC with Windows 2000. The PC is connected to the PowerPC board via a *background debug mode* interface [8].

Figures 8 and 9 show memory and time overhead comparisons, respectively, between the RCG algorithm, the ordinary incremental state saving (ISS) [10] and incremental state saving for only destructive instructions (ISSDI) [5]. The benchmark programs used are a Fibonacci number generator (FNG) with 100 iterations, a selection sort (SS) with 10 inputs, a 3 by 3 matrix multiplication (MM), and a random number generator (RNG) with 100 iterations. The results indicate that RCG algorithm achieves from 3.17X to 400X and from 2.5X to 300X reduction in memory overheads as compared to ISS and ISSDI, respectively (Figure 8). Furthermore, the RCG algorithm achieves an average of 5.7X and 4.1X reduction in execution time overheads when compared to ISS and ISSDI, respectively (Figure 9).

5. CONCLUSION

In this paper, we introduce a new reverse execution methodology for programs. To realize reverse execution, our methodology generates the reverse of a program by a static analysis at the assembly level. Our methodology is new because state saving can be largely avoided even with programs including many destructive instructions. This cuts down memory and time overheads introduced by state saving during forward execution of programs. Moreover, the methodology provides instruction by instruction reverse execution at the assembly instruction-level without ever requiring any forward execution of the program. In this way, a program can be run backwards to a state as close as one assembly instruction before the current state.

Since the generation of a reverse program is performed by the analysis at the assembly level, the methodology introduced in this paper provides reverse execution capability for programs without source code. Also, since both the forward code and the reverse code are executed in native machine instructions, these executions can be performed at the full speed of the underlying hardware.

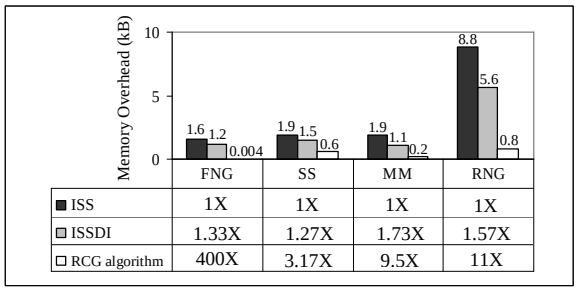


Figure 8: Memory overhead comparison.

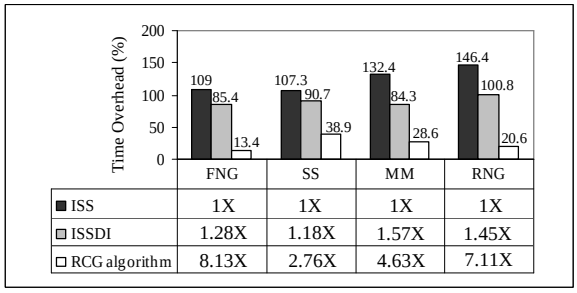


Figure 9: Time overhead comparison.

6. ACKNOWLEDGEMENTS

This research is funded by the State of Georgia under the Yamacraw initiative and by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, SUN and Synopsys. We also acknowledge helpful conversations with Dr. Santosh Pande.

7. REFERENCES

[1] H. Agrawal and J. Horgan. Dynamic program slicing. *SIGPLAN Notices*, 25(6):246–256, June 1990.

[2] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. Technical Report GIT-CC-02-49, Georgia Institute of Technology, September 2002.

[3] S. Bellenot. State skipping performance with the time warp operating system. In *Proceedings of the Sixth Workshop on Parallel and Distributed Simulation*, pages 53–64, 1992.

[4] M. R. Birch et al. Dynalab: A dynamic computer science laboratory infrastructure featuring program animation. *ACM SIGCSE Bulletin*, 27(1):29–33, March 1995.

[5] C. Carothers, K. Perumalla, and R. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3), July 1999.

[6] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with leonardo. *Journal of Visual Languages and Computing (JVLC)*, 11(2):125–150, April 2000.

[7] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in time warp simulation. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, pages 3–10, 1993.

[8] *MPC860 Users Manual*. Motorola Inc., 1998. http://e-www.motorola.com/brdata/PDFDB/docs/MPC860UM.pdf

[9] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[10] D. West and K. S. Panesar. Automatic incremental state saving. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, pages 78–85, 1996.