Instruction-level Reverse Execution for Debugging

Tankut Akgul and Vincent J. Mooney III

Technical Report GIT-CC-02-49
Georgia Institute of Technology

Abstract

Reverse execution provides a programmer with the ability to return a program to a previous state in its execution history. The ability to execute a program in reverse is advantageous for shortening software development time. Conventional techniques for reverse execution rely on saving a state into a record before the state is destroyed. State saving introduces both memory and time overheads during forward execution.

Our proposed method introduces a reverse execution methodology at the assembly instruction level with low memory and time overheads. The methodology generates from a program a reverse program by which a destroyed state is almost always regenerated rather than being restored from a record. This significantly reduces state saving.

The methodology has been implemented on a PowerPC processor with a custom made debugger. As compared to previous work all of which heavily use 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 reduction in execution time overhead.

1 Introduction

As human beings are quite prone to making mistakes, it is very difficult for a programmer to write an error-free program without going through a debugging cycle. For this reason, debugging is an important and inevitable part of software development.

Locating bugs by just looking at 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 the bugs 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 thus 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 report, a novel reverse execution methodology in software is described. The described 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 report, the word "instruction" refers to an assembly instruction.*

In Section 2, the main challenges of reverse execution are explained and the motivation behind this report is stated. In Section 3, related work is presented. In Sections 4 and 5, the approach of this report is introduced. The experimental results are presented in Section 6. Finally, Section 7 concludes the report.

2 Background and Motivation

An execution of a program T on a processor P can be represented by a transition among a series of processor states $S = (S_0,\ S_1,\ S_2,\ ...)$. From this representation, instruction-level reverse execution of a program can be defined as follows:

Definition 2.1 *Instruction-level Reverse Execution*: Reverse execution of a program T running on a processor P can be defined as taking P from its current state S_i to a previous state S_j $(0 \leq j < i)$ by executing a set of instructions which reverses the effect of instructions in T. The closest achievable distance between S_i and S_j *without any forward execution* determines the granularity of 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 instruction-level reverse execution. □

The simplest approach for obtaining a previously attained state is saving that state before it is destroyed. However, saving a state during execution of a program introduces two overheads: *memory* and *time*. A solution to reduce memory and time overheads would be

to decrease the frequency of state saving during program execution. However, this prevents an *immediate* return (i.e., a return without any forward execution) to an arbitrary point in execution history where state is not saved. Therefore, in applying state saving, there usually exists a tradeoff between the closest previous state that can be restored without any forward execution and memory/time overheads due to state saving.

Performance and memory constraints or lack of compiler support usually forces assembly language programming of some software components such as small scale embedded applications, firmware for consumer electronics, DSP libraries and operating system modules such as schedulers, high performance I/O routines or device drivers. For instance, the majority of boot code for the computer system of the Pathfinder Spacecraft was written in assembly language because it was critical for the computer to boot up very quickly in case of a failure [26]. Therefore, during debugging of such software components, programmers have to be involved in instruction-level program execution. Furthermore, in implementing a language construct such as a pointer to an integer, sometimes the compiler generates assembly different from what the programmer expected. These reasons are why most of the debugger tools for software contain assembly-level execution views. Thus, reverse execution at the instruction-level granularity turns out to be very helpful when debugging these sorts of software components.

During debugging of programs written either in a high-level or in a low-level programming language, programmers typically use a single-stepping facility to locate bugs. It is not an uncommon circumstance that programmers miss a bug location by executing just one more step over the next statement or instruction in the program. In such a case, instruction-level reverse execution provides an extremely fast backup capability.

However, due to the tradeoff between memory/time overheads and the closest previous state that can be restored, providing instruction-level state recovery by state saving can translate into very high memory and time overheads during execution of a program. Therefore, our goal is to achieve reverse execution at the native instruction level with low memory and time overheads, which will open the way for addition of a missing feature, instruction-level reverse execution, to state-of-the-art debuggers.

3 Related Work

Reverse execution has been researched in several contexts. In this section, we will mention previous work according to different application areas of reverse execution and also according to different techniques applied.

Zelkowitz provides a state restoration capability by inserting trace statements into the programming language [30]. Each trace statement includes an option which indicates either a condition or a label. Program state is captured starting from a trace statement until the condition indicated by the trace statement is satisfied or until the label indicated by the trace statement is reached. However, the programmer has to anticipate which parts of the program he or she might have to re-execute and then has to insert trace statements in those program parts beforehand.

Agrawal et al. provide a statement-level state restoration capability of a program written in a high-level programming language [2]. They statically associate with each assignment statement a set of variables, called a change-set, which is modified by that statement. Then,

during the execution, the associated variables in the change-set are recorded (saved to memory) for rollback. However, obviously, although this approach provides a statement-level state restoration capability, it might cause large memory and time overheads during program execution, especially with programs that modify the state frequently.

Reverse execution is also applied in so-called replay techniques for efficient debugging of nondeterministic sequential or parallel programs using either hardware [4, 25] or software [11, 20, 23, 24]. In a replay technique, first, the state of a program is saved at a coarser granularity during execution of the program and then the program state at a finer granularity is reconstructed by replaying the program using previously saved runtime information. In hardware approaches, state saving is handled by hardware with inflexibility and high cost but with little or no performance overhead. On the other hand, in software approaches, state saving is handled by software with flexibility and low cost but with high performance overhead. A typical drawback of these replay techniques is that since the recorded trace keeps only partial information about program state, execution can be restarted only at the beginning of a time interval in execution history but not at an arbitrary program point.

Reverse execution finds its application in a limited sense in the area of debugging optimized code as well [1, 17, 29]. Hennessy introduces the term "currency" of a variable. A variable is current at a program point if the value of the variable at that program point is the same as the variable's expected value which is deduced from the source code. Since code optimizations such as code motion and dead variable elimination may move or remove assignments to variables in the object code, the value of a variable at a certain point in the optimized code may not be equal to the value of the variable at the corresponding point in the unoptimized code, which causes the variable to be "noncurrent" at that program point. In such a case, the current value of the variable has to be recovered to provide the user with a consistent view of the program being debugged. This recovery operation is where reverse execution comes into play. A typical recovery technique in this field is to reevaluate noncurrent variables using appropriate definitions of those variables in the program. However, since the main focus in this area has been on the determination of whether a variable is current or not at a program point rather than on the recovery of a noncurrent variable, the recovery techniques applied in this area are generally very restrictive and ineffective. For instance, Wismuller reports that only 2-5% of all encountered noncurrent variables can be recovered in his benchmarks [29].

Floyd makes use of reverse execution or backtracking approach in the area of nondeterministic algorithms [13]. A nondeterministic algorithm is an algorithm which may come up with different solutions to a problem at each run of the algorithm. However, the solution is not reached by a random process but by intelligently and incrementally constructing a right path which leads to a success. In Floyd's approach, whenever a nondeterministic algorithm enters a path leading to a dead end, the algorithm state at the most recent point where a decision is made is restored and alternative solutions are sought from that point on. In this way, all possible solutions out of a nondeterministic algorithm can be obtained, which essentially converts a nondeterministic algorithm into a deterministic one. This technique turns out to be very useful for theorem proving in artificial intelligence as well. Floyd achieves state restoration by defining a reverse operation for each operation in a nondeterministic al-

gorithm. However, except for constructive operations such as "x = x + 1," reverse operations are realized by applying state saving.

Reverse execution is also used in computer science education where students can easily navigate back and forth through well-known algorithms to understand the behavior of such algorithms. For this purpose, the common technique applied is program animation [6, 9]. Program animation constructs a virtual machine with a reversible set of instructions. Since these instructions are reversible, the program can be run backwards. However, in program animation, a program is usually interpreted, which slows down the animation considerably, and makes it impossible to execute the program using native machine instructions, 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.

Two other application areas of reverse execution are optimistic or speculative computation [14, 15, 18] and fault tolerance [8, 19]. A computation is optimistic if incorrect computation is allowed during execution. In parallel executions, tasks usually have to block due to synchronization requirements on shared data. In optimistic parallel executions, blocking of the tasks on shared data is prevented and the tasks are allowed to execute independently, which potentially improves the execution performance but at the same time allows incorrect computation. Then, errors caused by possible incorrect computations are recovered by rolling back the computation of erroneous tasks to a point in time where state is known to be correct. Similarly, reverse execution for fault tolerance is performed by rolling back in case software errors occur, which is usually seen in places such as database transaction systems [5, 16].

Rolling back computations or transactions is usually achieved by periodic or incremental state saving. In periodic state saving [12], the whole processor state is recorded periodically at certain checkpoints during simulation. Then, a previous state at a checkpoint can be recovered by restoring that state from the record. However, in this method, a previous state at an arbitrary point that is not a checkpoint cannot be immediately recovered, which results in a coarser granularity reverse execution. If the checkpointing interval is reduced to provide a finer granularity reverse execution, memory and time overheads of state saving are increased. Moreover, recording the whole processor state at each checkpoint causes redundancy because some portion of the processor state may be kept unchanged throughout several checkpoints. In incremental state saving [28], instead of recording the whole processor state, only the modified parts of a state are recorded. However, in programs where the modified state space is large, memory and time overheads of incremental state saving might again exceed affordable limits.

Carothers et al. introduce another approach for optimistic parallel simulations [7]. This approach is source transformation. 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 granularity.

In the next section, we will give an overview of how we achieve an instruction-level reverse execution of a program under consideration. Then, the details of our approach will be explained in Section 5.

4 Overview of our Approach

Our approach is mainly based on regenerating a previously destroyed state rather than restoring the state from a record. When state regeneration is not possible, however, we recover a destroyed state by state saving. Therefore, our solution is a hybrid solution between state regeneration and state saving. In this section, we will explain how we achieve state regeneration. We will also describe our state saving method in Section 4.3.

Suppose that an execution of a program T on a processor P causes P to attain the series of states $S = (S_0,\ S_1,\ S_2,\ ...)$ where the distance between two consecutive states is one instruction. Now, assume that we can generate another program RT, the reverse of T, such that when a specific portion of RT is executed in place of T when P is at a state $S_i = (PC_i, M_i, R_i)$, the state of P can be brought to a previous state $S_j = (PC_j, M_j, R_j)$ $(0 \leq j < i)$. In other words, RT recovers a previously destroyed state. Then, the execution of T can be reversed by executing RT in place of T. However, practically, it might be hard to implement such a program RT. This is due to the following reasons:

(1) Typically, processors include auxiliary hardware usually not accessible by the instructions directly. The processors usually manipulate this kind of hardware implicitly. Therefore, it is typically hard to recover indirectly modified state in this kind of hardware. As an example, consider the overflow register of a processor. The overflow register is written indirectly by an operation such as "$c = a + b$" if an overflow occurs during such an operation. However, many processors do not specify an instruction to directly write to the overflow register which makes it hard to recover the overflow register.

(2) Generally, writing a value to the program counter either by a branch instruction or by direct modification causes an immediate jump to the location designated by the written value. Therefore, as soon as RT were to recover the program counter, the execution of RT would immediately be broken. This suggests that the program counter should be recovered only at the end of the execution of a specific portion of RT and just before the user switches back to forward execution. However, since it is not known a priori what program part the user will reverse execute (i.e., which portion of RT the user will run), it is impractical to recover the program counter inside RT.

(3) If an instruction modifies a memory location, the instruction encoding only tells us the modified address but not the physical location actually being modified in the memory hierarchy (i.e., L1 cache, L2 cache or main memory). Without the knowledge of the physical location actually being modified, it is typically hard to recover the exact physical memory state.

Therefore, let us define a processor state $S' = (M', R')$ which excludes the program counter (PC) value and which includes only directly modified memory (M') and directly

modified register (R') values (i.e., M' and R' only include the memory locations and registers that appear as operands of the instructions of T). Moreover, let us define M' to interchangeably represent either processor cache or main memory, whichever is used for keeping a program value at a certain time.

Given our definition of state S', we now introduce an *instruction-level reverse program* as follows:

Definition 4.1 *Instruction-level Reverse Program*: Suppose that a processor P attains the series of states $S' = (S'_0,\ S'_1,\ S'_2,\ ...)$ during its execution of a program T where between a state $S'_i \in S'$ and the preceding state $S'_{i-1} \in S'$, there exists only one instruction that directly modifies a memory or a register value. Now, suppose that another program RT exists such that when a specific portion of RT is executed in place of T when P is at a state $S'_i = (M'_i, R'_i)$, the state of P can be brought to a previous state $S'_j = (M'_j, R'_j)$ $(0 \leq j < i)$. If RT contains an executable portion for changing the state of P from any state $S'_i \in S'$ to any other previous state $S'_j \in S'$ $(j < i)$ for any possible state sequence S' during execution of T, then RT is called the instruction-level reverse program of T. □

Assuming that we can generate an instruction-level reverse program RT of T, we can recover all memory and register values that are directly modified by T for every possible execution of T. However, since the program counter value carries important debugging information, we still have to provide a means for restoring the program counter value. We solve this problem by leaving the recovery of the program counter value to the debugger tool. The debugger tool associates the address of each instruction in T with the beginning address of the corresponding portion in RT which reverses the effect of that instruction. In this way, when a part of T is reverse executed by executing the corresponding portion in RT, the debugger tool restores the value of the program counter by using the connection between the addresses in T and RT. Similarly, we handle the recovery of indirectly modified memory/register values which have an effect on T's state by the help of the debugger tool. For more information about recovering indirectly modified memory/register values, please refer to the Appendix.

In order to be able to generate an instruction-level reverse program RT for a program T running on a processor P, we should first relate the states in a particular sequence S' attained by P to the instructions in T.

Definition 4.2 *The Relation of a State Sequence to an Instruction Sequence*: The state sequence $S' = (S'_0,\ S'_1,\ S'_2,\ ...S'_n)$ during an execution of a program T on a processor P can be associated with a set of instructions in T which completes in a sequence $I = (\alpha_1,\ \alpha_2,\ ...\ \alpha_n)$ where $\alpha_i \in I$ changes the state of P from $S'_{i-1} \in S'$ to $S'_i \in S'$. Note that since a state $S'_i \in S'$ includes neither the program counter value nor indirectly modified memory and register values, the sequence I does not contain any branch instructions (which modify the program counter value) or the instructions that only indirectly modify memory or register values. □

Now, we will define another term, *reverse instruction group*, as follows:

Definition 4.3 *Reverse Instruction Group* (*RIG*): Suppose that one could generate a group of one or more instructions denoted by RIG_i for an instruction $\alpha_i \in I$ such that if RIG_i is executed with

P being at state $S'_i \in S'$, the state of P can be brought back to state $S'_{i-1} \in S'$. In other words, RIG_i can undo the effect of α_i on P's state. We state that RIG_i is a group consisting of one or more instructions because multiple instructions may be needed to reverse the effect of α_i. □

Then, the effect of the complete sequence I in Definition 4.2 can be reversed by executing the corresponding RIGs in an order opposite to the completion order of I, that is, by generating a sequence such as $I_{RIG} = (RIG_n, RIG_{n-1}, ..., RIG_1)$ where a reverse instruction group $RIG_i \in I_{RIG}$ reverses the effect of $\alpha_i \in I$ $(1 \leq i \leq n)$.

Therefore, in this report, we introduce a static algorithm, the *reverse code generation* (*RCG*) algorithm, which generates a RIG for each instruction (excluding the branch instructions and the instructions that only indirectly modify memory and register values) in a program T and combines the generated RIGs to make these RIGs complete in an order opposite to the completion order of the instructions in T. Since the instruction sequence I that may result from an execution of T may vary according to dynamic control flow of T, the RCG algorithm combines the RIGs by binding the RIG sequence to be executed during reverse execution to the dynamic control flow information of T.

Since inter-procedural control flow information is hard to capture statically (e.g., due to indirect function calls), the RCG algorithm is mainly intra-procedural. That is, the RCG algorithm combines the RIGs to generate the reverse versions of the procedures/functions in a program rather than to generate the instruction-level reverse program directly. Then, the RCG algorithm combines the reverse versions of the procedures/functions by a glue code which may employ state saving (see Section 4.9).

Note, however, that a perfect separation of a procedure/function F from other procedures/functions within a program may not always be possible because there may be calls to other procedures/functions within the body of F. Therefore, in such a case, the RCG algorithm first divides F into sub-procedures/sub-functions at the assembly level which are separated from each other according to calls to other procedures/functions within F. Then, each sub-procedure/sub-function is treated as if it were a standalone procedure/function. We call these sub-procedures/sub-functions *program partitions*.

Listing 1 shows the pseudo code illustrating the main function of the RCG algorithm. The RCG algorithm first calls a function, *Init_RCG()*, which generates program partitions from a program and prepares other necessary data structures (line 1 of Listing 1). Then, the RCG algorithm enters a main loop (line 2) where it analyzes each program partition assembly instruction by assembly instruction in the order the instructions are placed by the compiler (lexical order). After an instruction is read, the RCG algorithm executes a function, *Find_CF()*, which gradually obtains the intra-partitional control flow information while the program is being scanned (line 6). Then, if the instruction that has been read directly modifies a memory or a register value, the RCG algorithm checks whether the instruction is inside a loop. If the instruction is not inside a loop, the RCG algorithm directly calls a function named *Gen_RIG()* (line 11). *Gen_RIG()* is responsible for the generation of a RIG for the instruction under consideration. On the other hand, if the instruction is inside a loop, the RIG generation for the instruction may require special handling which is performed by *Loop_Gen()* function called at line 9. *Loop_Gen()* which will be explained in Section 4.7.1 basically calls *Gen_RIG()*; however, *Gen_RIG()* may not generate a complete RIG in a single

Listing 1 The main function of the RCG algorithm

```
Input: A program T
Output: An instruction-level reverse program RT for T
begin
 1 Init_RCG()
 2 for all program partition F_i ∈ T do
 3   cur_instr = address of the first instruction of F_i
 4   while there are unread instructions in F_i do
 5     α = Read_instruction(cur_instr) /*read the instruction pointed to by cur_instr*/
 6     Find_CF()
 7     if α directly modifies a memory location or a register then
 8       if α is in a loop L then
 9         RIG_α = Loop_Gen(α, iteration(L))
10       else
11         RIG_α = Gen_RIG(α)
12       end if
13       if RIG_α is complete then
14         Combine_RIGs(RIG_α)
15       end if
16     end if
17     if (α is in a loop L) ∧ (end of L is reached) then
18       if (L requires another traversal) then
19         cur_instr = address of the first instruction of L
20       else
21         cur_instr = address of the next instruction in F_i
22       end if
23     else
24       cur_instr = address of the next instruction in F_i
25     end if
26   end while
27   Combine_Partitions()
28 end for
end
```

traversal of the loop in which the instruction resides. Therefore, *Loop_Gen()* ensures the completion of a RIG for the instruction under consideration by requesting multiple traversals over the loop body (see Section 4.7.1). Finally, if a complete RIG is generated, another function, *Combine_RIGs()*, combines the generated RIG with the previously generated RIGs. (line 14). At the end of the main loop, when the reverse version of the program partition that is currently being analyzed is completed, the RCG algorithm connects the reverse version of the current program partition to the reverse versions of the previously analyzed program partitions by calling a function named *Combine_Partitions()* (line 27).

In the following sections, we will describe the functions that are called by the main function of the RCG algorithm.

4.1 *Init_RCG()*: Building the initial data structures of the RCG algorithm

Since the RCG algorithm operates on each program partition separately, the first thing to do is to determine the program partitions from the instructions of the program under consideration.

The RCG algorithm determines the program partitions in a program by constructing a *partitioned control flow graph*, $PCFG$=(N,E,**start**,**exit**), for each program partition in the program. N is the set of nodes, E is the set of edges representing the flow of control between the nodes, and **start** and **exit** are the unique entry and exit nodes of a PCFG, respectively. Each node in a PCFG represents a *basic block* (*BB*). A basic block is a single entry, single exit block of a maximal number of consecutive instructions. Since a PCFG construction is performed over assembly instructions, a BB in a PCFG 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., we assume that a multi-way 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).

Listing 2 *Init_RCG()*: Building the initial data structures of the RCG algorithm

Input: A program T
Output: The PCFGs for the program partitions in T and the CG for T
begin

```
 1 i = 0
 2 repeat
 3    PCFG_i = φ /*initialize PCFG_i to be empty*/
 4    PCFG_i += start block
 5    Label_Edges(start block)
 6    repeat
 7       α = Read_the_next_instruction()
 8       if end of current BB is reached then
 9          Add current BB to PCFG_i
10          Label_Edges(current BB)
11       end if
12    until (α = "call") or (α = "return")
13    PCFG_i += exit block
14    Grow_CG()
15    i = i+1
16 until end of the program is reached
```

end

Listing 2 shows the pseudo code for the function *Init_RCG()*. *Init_RCG()* builds a PCFG for each program partition in the program under consideration by reading the instructions of the program in a loop (lines 6 to 12 of Listing 2). *Init_RCG()* starts the construction of $PCFG_i$ by inserting a **start** block at the beginning of $PCFG_i$ (line 4) and by calling a function *Label_edges()* which labels the outgoing forward edge of the **start** block (line 5). Edge-labeling, which will be explained in Section 5.1, is performed to assist in the determination of *intra-partitional* control flow information and the generation of the RIGs. Then, in the loop, *Init_RCG()* adds BBs to $PCFG_i$ until a "call" instruction (i.e., an instruction that

is used to call a procedure/function) or a "return" instruction (i.e., an instruction that is used to return from a procedure/function call) is encountered in the program being analyzed. After a new BB is added to $PCFG_i$, *Init_RCG()* calls *Label_edges()* to label the outgoing forward edges of the newly added BB (line 10). When *Init_RCG()* encounters a "call" or a "return" instruction in the program being analyzed, *Init_RCG()* ends the construction of $PCFG_i$ by adding an **exit** block to the end of $PCFG_i$ (line 13). The instruction just after the call or the return instruction, on the other hand, starts a new program partition and thus a new PCFG. After a PCFG is constructed, *Init_RCG()* calls *Grow_CG()* which gradually constructs another directed graph, a *call graph* (*CG*), for the program under consideration (line 14). The CG is used for obtaining the *inter-partitional* control flow information and will be explained in Section 4.9.

4.2 *Find_CF()*: Finding the intra-partitional control flow information

In this section, we give an overview of the function *Find_CF()* (called from line 6 of Listing 1), namely, we will outline how the RCG algorithm obtains the control flow information of a program partition under consideration.

As explained in Section 4.1, each node in the PCFG of a program partition designates a BB. The important property of a BB is that the instructions within a BB always complete in lexical order. In other words, the completion order of the instructions within the BBs is not dependent on any condition. This lack of dependence automatically fixes the ordering of the corresponding RIGs in the reverse code. Therefore, the PCFG construction reduces the needed intra-partitional control flow information to the information of control flow among the BBs of a program partition only.

A confluence point of edges encountered in the PCFG of a program partition is the only point where a decision has to be made about the control flow during reverse execution. Therefore, the only information required about the control flow between the BBs of a program partition is the information which reveals under which condition a confluence point (or a join point) in a PCFG is dynamically reached along a particular incoming edge to that confluence point.

The incoming edge along which a confluence point is dynamically reached is decided by the control flow predicates that are associated with each incoming edge to that confluence point. Let us illustrate this with the following example.

Example 4.1 Consider the function foo shown in Figure 1(a) that is written in the C programming language. The assembly listing (for the PowerPC 860) and the PCFG of foo are shown in Figure 1(b) and Figure 1(c), respectively. When the RCG analysis arrives at point **P** shown in the figure, it is necessary to know along which incoming edge BB4 will dynamically be reached in order for the RCG algorithm to generate the appropriate branch instructions which will reverse execute foo backwards from **P**. The edge to be taken to reach BB4 is decided by the conditional branch instruction at the end of BB1 which causes the flow of control to be divided into two separate paths before reaching **P**. The predicate expression of this conditional branch instruction is shown as $r_{10} > 100$ in Figure 1(c). If the value of the predicate expression $r_{10} > 100$ is true, then **P** is reached along one incoming edge (from

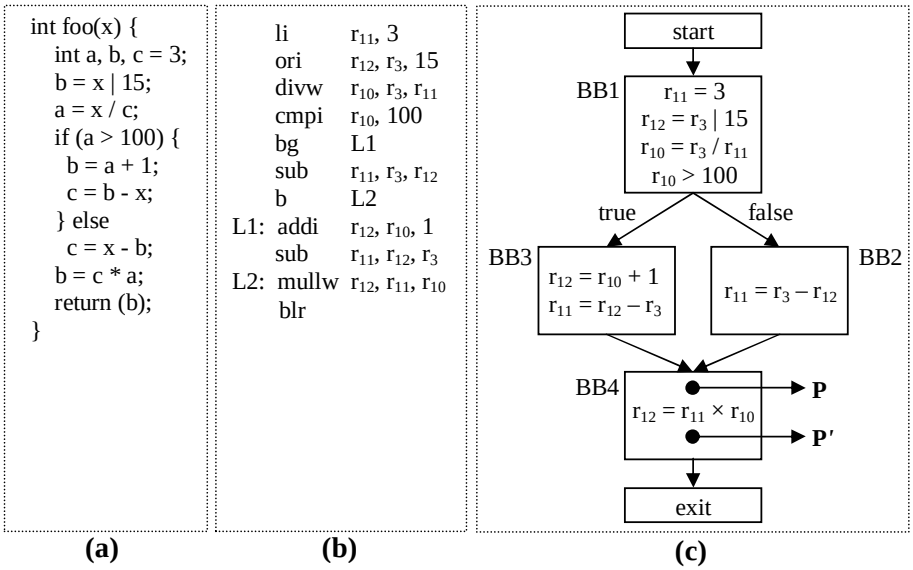


Figure 1: (a) A simple program in C. (b) Corresponding assembly instructions. (c) Corresponding PCFG.

BB3); however, if the value of the complementary predicate expression $r_{10} \leq 100$ is true, then **P** is reached along the other incoming edge (from BB2). □

Therefore, to determine the dynamically taken incoming edge to a confluence point P, one needs to find the following two items:

(1) The predicate expression Υ_i associated with each incoming edge e_i to P such that when the value of Υ_i for an edge e_i becomes true, that edge is taken to reach the confluence point P. Here, the index i varies from one to n where n is the number of incoming edges to point P.

(2) The predicate expression Υ_{true}, among the predicate expressions found in (1), that becomes *true* during a specific iteration or execution arriving at the confluence point P.

To find (1), *Find_CF()* uses special labels assigned to the edges of the PCFG describing the program partition under consideration. As will briefly be mentioned in Section 4.3 shortly and then will be described in more detail in Section 5.3.2, edge labels also assist in finding reaching definitions which are essential for RIG generation. Note that we could also have used a standard *control dependency graph* (*CDG*) [22] analysis to determine the predicate expressions; however, due to the desire to detect the predicate expressions and reaching definitions together in an efficient way, edge-labeling is preferred over a CDG analysis. We introduce the edge-labeling algorithm and then describe the predicate expression determination in Sections 5.1 and 5.2, respectively.

To find (2), we follow two possible methods. The first method is to save the predicate values during forward execution of a program partition. In this first method, it is sufficient to save the values of $n-1$ of the n predicate expressions found in (1). Because if none of the $n-1$ predicate expressions happen to be true, then the remaining n^{th} predicate expression

is guaranteed to be true. For instance, in Example 4.1, since there are only two predicate expressions (namely, $r_{10} > 100$ and $r_{10} \leq 100$) that are associated with the two incoming edges to point **P**, saving the value of only one of the predicate expressions is sufficient to determine which edge is taken to reach **P**. The drawback of this method is that, obviously, state saving of predicate values causes some memory and time overheads during forward execution of a program partition.

The second method to find (2) is to reevaluate the predicate expressions during reverse execution. In this second method, it is again sufficient to reevaluate the values of $n-1$ of the n predicate expressions found in (1) due to the same reason given in the explanation of the first method. As an example for the second method, if the predicate expression $r_{10} > 100$ – see Figure 1(c) – is chosen to be reevaluated, the value of $r_{10} > 100$ can be found once again at point **P** by executing the compare instruction "*cmpi* r_{10}, 100" during reverse execution. In this second method, there is no time nor memory overhead encountered during forward execution of the program partition under consideration. However, if the value of any variable used in a predicate expression to be reevaluated (e.g., the value of r_{10} in the expression $r_{10} > 100$) has already been destroyed before reaching the reevaluation point, then that destroyed value must be recovered during reverse execution before the predicate expression can be reevaluated. This requirement may cause a slower reverse code to be generated as compared to the code generated by using the first method. The amount of possible performance degradation of the reverse code depends on how many destroyed variables need to be recovered in order to reevaluate the predicate expression under consideration.

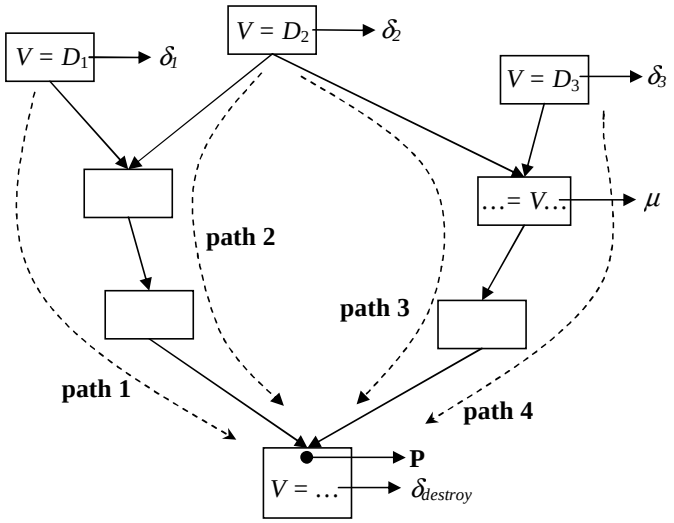
Both methods explained in the previous two paragraphs are equally applicable. Since our primary concern is to reduce memory and time overheads of forward execution, the second method seems to be more preferable in most cases over the first method. Therefore, we use the second method as a default method. However, we still provide the programmer with an option that minimizes a cost function whose main parameters are memory and time overheads of forward execution and the speed of reverse execution.

4.3 *Gen_RIG()*: Generating a reverse instruction group

Gen_RIG() function (called from lines 9 and 11 of Listing 1) involves the generation of a RIG for every instruction that directly modifies a register or a memory location in a program partition.

Suppose that a definition $\delta_{destroy}$ destroys the value D of a variable (a directly modified register or memory location) V at a program partition point as shown in Figure 2. Let us name the program point just before $\delta_{destroy}$ as **P**. To recover D, one needs to know at what point in the program partition D might be assigned to V. This is exactly the same problem as finding the definitions of V reaching point **P**.

Gen_RIG() follows a more efficient technique than the common technique of using bit-vectors to determine reaching definitions at a program partition point [22]. The main reason for the increased efficiency is that *Gen_RIG()* does not require an iterative solution of data-flow equations. First, *Gen_RIG()* employs a method called *value renaming* which refers to giving a different name to every definition of a directly modified register or memory location. Value renaming is same as the renaming operation in the well-known *static single assignment* (*SSA*) form generation [10]. By value renaming, different definitions of a variable can easily



$D \in \{D_1, D_2, D_3\}$
Re-executing δ_1 recovers D for path 1
Re-executing δ_2 recovers D for paths 2 and 3
Re-executing δ_3 recovers D for path 4
Extracting V out of μ recovers D for paths 3 and 4

Figure 2: Recovering a destroyed variable.

be distinguished from one another. Then, *Gen_RIG()* uses the labels on the edges of the PCFG to efficiently find reaching definitions at each program partition point. The details of how value renaming is performed and how reaching definitions are determined will be described in Sections 5.3.1 and 5.3.2, respectively.

Each statically reaching definition δ_i of V at point **P** might correspond to the instance where D is actually assigned to V (Figure 2). The definition that corresponds to the actual assignment instance is the definition that dynamically reaches point **P**. Therefore, recovering D means recovering the definition of V that dynamically reaches point **P**.

The definition of V that dynamically reaches point **P** depends on the dynamically taken path to **P**. However, the path that will actually be taken is typically not known prior to program execution. Therefore, we use the following technique to recover D: we generate sets, each of one or more instructions, where each set recovers one or more definitions of V statically reaching **P** along at least one path. For instance, referring to Figure 2, we can generate a set which recovers δ_1. This set indeed recovers D if **path 1** is dynamically taken. Similarly, we can generate another set which recovers δ_2. This second set indeed recovers D if either **path 2** or **path 3** is dynamically taken. We generate as many sets as necessary to cover all possible paths to $\delta_{destroy}$ from the definitions of V reaching **P**. If more than one set is generated, we tie the sets together via conditional branch instructions. The predicates of the conditional branch instructions carry the dynamic control flow information of the program as outlined in Section 4.2. Therefore, the correct set to be executed during reverse execution is automatically selected by these predicates. If a predicate is also destroyed before $\delta_{destroy}$, then, in the same way, we generate the sets which recover that predicate. The sets that recover the reaching definitions of V, the conditional branch instructions (if any) that

are used to gate these sets and the instructions (if any) that are generated to recover the predicates all together constitute a RIG for $\delta_{destroy}$.

Let us now describe how a set of instructions which we will denote by ζ can be generated to recover at least one definition of V reaching $\mathbf{P}$. There are three techniques that are followed to generate a ζ: the *redefine technique*, the *extract-from-use* technique and the *state saving* technique.

4.4 The redefine technique

The redefine technique is to put into ζ the instruction α_i which computes D_i at the definition site δ_i statically reaching point $\mathbf{P}$(Figure 2). If any one of the variables that is used for computing D_i is also destroyed, then the instruction which recovers that variable must be inserted before α_i in ζ and this must be applied recursively for all other modified variables in the dependency chain.

The redefine technique can potentially recover only one definition of V reaching $\mathbf{P}$: the definition it redefines. Note that, however, the external value of an input variable (e.g., a global variable or an input argument) of a partition is certainly not defined within the partition but comes from outside of the partition. Therefore, the external values of a partition cannot be recovered by the redefine technique.

The following example illustrates how the redefine technique works.

Example 4.2 *The redefine technique*: Consider the instruction which overwrites the value of register r_{12} in BB4 in Figure 3 (we need the overwritten value of r_{12} because the overwritten value is used both in BB2 and BB3). Let us name this instruction as α and the analysis points just before α and just after α as $\mathbf{P}$ and $\mathbf{P}'$, respectively. There are two different definitions of r_{12} reaching $\mathbf{P}$ on two different paths: "$r_{12} = r_{10} + 1$" and "$r_{12} = r_3 \mid 15$". Therefore, the value of r_{12} at point $\mathbf{P}$ is either "$r_{10} + 1$" or "$r_3 \mid 15$". Moreover, neither r_{10} nor r_3 is modified after being used to define r_{12} and before point $\mathbf{P}'$. Therefore, the destroyed value of r_{12} can be recovered on one path by executing the set "$r_{12} = r_{10} + 1$", and it can be recovered on the other path by executing the set "$r_{12} = r_3 \mid 15$". □

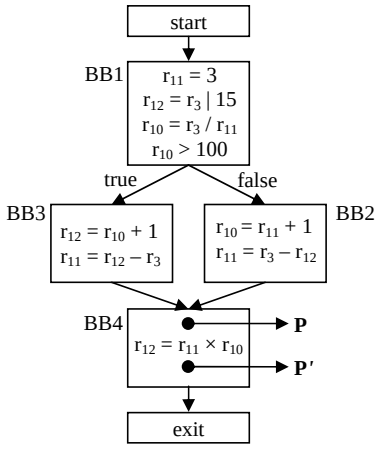


Figure 3: An example PCFG.

4.5 The extract-from-use technique

The extract-from-use technique is to put into ζ an instruction β which extracts the destroyed value of V out of a use μ (including a possible use of V by $\delta_{destroy}$ itself) on the path(s) between $\delta_{destroy}$ and any definition of V reaching $\mathbf{P}$ (Figure 2). However, again, if any other variable in β which is used for extracting V is also destroyed, then an instruction which recovers that variable must be inserted before β in ζ and this must be applied recursively for all other modified variables in the dependency chain.

As opposed to the redefine technique, the extract-from-use technique can recover multiple definitions (δ_2 and δ_3 in Figure 2) of V reaching $\mathbf{P}$. Moreover, since the external value of an input variable of a partition may be used within the partition, the input values to a partition might still be recovered by using the extract-from-use technique. However, the extract-from-use technique is less likely to be applicable than the redefine technique because there might not always be a use μ on a path to $\delta_{destroy}$, and, even if a use is available, μ's operation might not always allow such an extraction of the value of V. For example, the instruction "$r_3 = r_1 \ / \ r_2$" might prevent the extraction of r_1 or r_2 since the result of the division operation might be truncated due to the limited precision r_3 can represent. In general, operations such as "integer add", "integer subtract" and "integer multiply" allow extraction of values provided that the result of any of these operations is not truncated due to an overflow/underflow. A "shift" operation is also reversible if bits are not lost due to a shift-out. On the other hand, operations such as "integer divide" and all other floating point calculations do not allow extraction of values due to a possible loss of precision on the result. The decision on whether or not to use the extract-from-use technique on the operations that might not be reversible in special situations such as overflow/underflow or shift-out is left to the programmer. For example, the programmer may use compiler warnings, the overflow/underflow detection logic of the processor or overflow/underflow detection code to ensure that the program is free of overflow/underflow situations.

The following example illustrates how the extract-from-use technique works.

Example 4.3 *The extract-from-use technique*: Consider again the instruction which overwrites the value of register r_{12} in BB4 in Figure 3. After the two definitions of r_{12} reaching $\mathbf{P}$, there are two uses of r_{12} on each path: "$r_{11} = r_{12} - r_3$" and "$r_{11} = r_3 - r_{12}$". Moreover, neither r_{11} nor r_3 is modified between the points of uses and point $\mathbf{P}'$. These subtractions are performed as integer operations and thus they are reversible provided that their results are not truncated. Thus, if the point $\mathbf{P}'$ is reached passing through the use "$r_{11} = r_{12} - r_3$", the destroyed value of r_{12} can be obtained by executing the set "$r_{12} = r_{11} + r_3$"; if $\mathbf{P}'$ is reached passing through the use "$r_{11} = r_3 - r_{12}$", then the destroyed value of r_{12} can be obtained by executing the set "$r_{12} = r_3 - r_{11}$". □

4.6 The state saving technique

The RCG algorithm applies the redefine and the extract-from-use techniques in a combination to come up with the smallest RIG. However, due to the limitations of these techniques described in the previous subsections, we may not be able to generate all of the sets necessary to cover all paths to $\delta_{destroy}$ (Figure 2). Even worse, as in the case of memory aliasing which will be described in Section 5.3.1, we may not be able to find the statically reaching

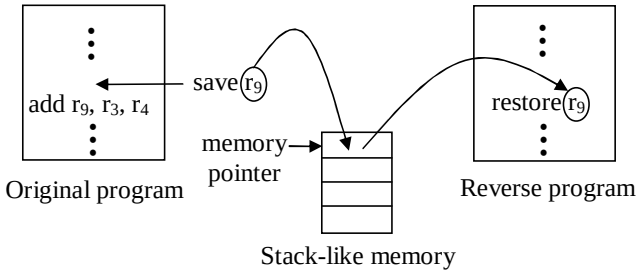


Figure 4: A diagram which illustrates the state saving method of the RCG algorithm.

definitions of V at all. In such circumstances, the RCG algorithm resorts to the state saving technique which is always applicable.

In general, we save a state by inserting a push-like instruction into the original code just before $\delta_{destroy}$. The inserted instruction saves the state (e.g., r_9 in Figure 4) that is being modified by $\delta_{destroy}$ into a free memory location that is pointed to by a memory pointer (usually a register) and moves the memory pointer to the next free location. Then, in the reverse program, a pop-like instruction is generated which moves the memory pointer to the next value to be restored and restores the saved value from memory.

A push-/pop-like instruction refers to an instruction which works in the same way as an ordinary push/pop instruction; however, a push-/pop-like instruction can work on any memory pointer, while a push/pop instruction can work only on the stack pointer. For instance, PowerPC 860 provides store-update and load-update instructions which can be used as push-like and pop-like instructions, respectively. Ordinary push and pop instructions are not considered for state saving in order to not possibly corrupt the stack. If the target architecture does not support pop-like/push-like instructions which automatically increment/decrement a memory pointer, save and restore operations are handled by using ordinary store and load instructions and by incrementing/decrementing a dedicated memory pointer explicitly.

4.7 An example of RIG generation

In the previous three subsections, we explained the three methods we use to generate a set which recovers at least one definition of the variable under consideration. We also stated that a RIG is nothing but a combination of those sets which cover all possible paths to the destruction point. In this section, we give an example of a complete RIG generation by using the PCFG shown in Figure 3.

Example 4.4 *RIG generation*: In Examples 4.2 and 4.3, we gave 4 different sets each of which recover the value of r_{12} along a particular path. Let us now use some of these sets to generate a complete RIG for recovering the value of r_{12}. Let us pick "$r_{12} = r_{10} + 1$" to recover r_{12} along the left path to point **P** and "$r_{12} = r_3 - r_{11}$" to recover r_{12} along the right path to point **P** in Figure 3. Since all paths are covered, these two sets are enough to generate a RIG. We should now combine these two sets by using a conditional branch instruction which determines along which path **P** is reached. The predicate of this conditional branch instruction is $r_{10} > 100$. However, we cannot use this predicate as it is because the value of r_{10} is destroyed in BB2. Therefore, we should first recover r_{10}. We can recover r_{10} by two successive applications of the redefine technique: we first redefine "$r_{11} = 3$" and

then redefine "$r_{10} = r_3/r_{11}$" (r_{11} is redefined because it is destroyed as well). Note, however, that since our aim is to recover r_{12} only, we should use a temporary register r_t instead of r_{11} and r_{10} not to destroy the values of r_{11} and r_{10} at point P. Therefore, a RIG for recovering r_{12} can be generated as follows:

```
        li      r_t, 3
        divw    r_t, r_3, r_t
        cmpwi   r_t, 100
        ble     L1
        addi    r_12, r_10, 1
        b       L2
L1:     sub     r_12, r_3, r_11
L2:     ...
```

□

Listing 3 *Gen_RIG()*: Generate a RIG

Input: An instruction α
Output: A RIG, RIG_α, for α
begin

```
 1 RIG_α = φ
 2 for all t = a register/memory location directly modified by α do
 3   C = ∞
 4   D = Find_Reaching_Defs(t, α)
 5   if all definitions are statically known then
 6     P = Paths(D,t)
 7     U = the set of uses of t (with reversible operators) along the paths in P
 8     M = set of all subsets (combinations of elements) of U
 9     for all Z ∈ M do
10       RIG_t = φ
11       P_s = Cover(Z)
12       for all μ_i ∈ Z do
13         RIG_t = Extract_from_use(t, μ_i, RIG_t)
14       end for
15       for all path p_j ∈ P − P_s do
16         RIG_t += Redefine(t, p_j, RIG_t)
17       end for
18       if sizeof(RIG_t) < C then
19         RIG_m = RIG_t
20         C = sizeof(RIG_t)
21       end if
22     end for
23     if C == ∞ then
24       RIG_m = State_save(α)
25     end if
26   else
27     RIG_m = State_save(α)
28   end if
29   RIG_α += RIG_m
30 end for
end
```

Listing 3 shows the pseudo code for the generation of a RIG (with minimum size cost, C) for an instruction α. To find the minimum cost RIG, we apply the extract-from-use technique (line 13) and the redefine technique (line 16) for different paths starting from reaching definitions of t (if a reaching definition cannot be statically found, we save state – line 27). For this purpose, we process all different uses (with reversible operators) and/or definitions on different paths, where each use/definition covers a set of one or more paths. If the cost of the final RIG is infinity, which means neither the extract-from-use technique nor the redefine technique can recover t, we apply the state saving technique (line 24).

4.7.1 Handling loops

As mentioned before, reverse code generation for a loop requires additional passes over the loop body for the recovery of some instructions within the loop. This section explains the reason behind multiple traversals.

A variable modified by an instruction α within a loop L may show a transient behavior at the early iterations of L until the values that come from outside of L are propagated into the loop body. Thus, the code that reverses the effect of α may be different for different instances of α (i.e., for the instances due to different loop iterations). Consider the following example.

Example 4.5 Figure 5 shows a loop with four instructions. The values obtained by the target operands of the instructions at successive iterations of the loop are also shown in the figure. As seen from the figure, it requires three loop iterations until a pattern is observed in the values obtained by the target operand of the first instruction. The values obtained by this operand are affected by the values input to the loop and are totally unrelated at the early instances of the first instruction in the loop. Therefore, it is necessary to reverse the effect of each such instance of the first instruction separately. □

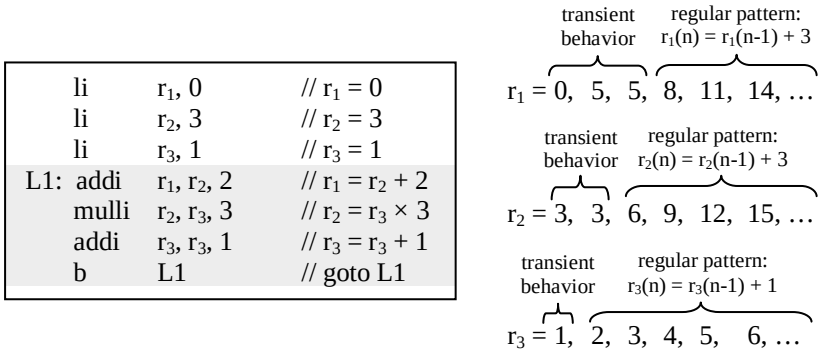


Figure 5: A simple loop.

Listing 4 shows the algorithm snippet for reverse code generation within loops. In order to capture the transient behavior explained, *Loop_Gen()* calls *Gen_RIG()* at each traversal of L for a single instance of α (line 1 of Listing 5). In other words, at the first traversal, *Loop_Gen()* generates a set of instructions, ζ_1, which reverses the effect of the first instance of α; at the second traversal, it generates another set of instructions, ζ_2, which reverses the effect of the second instance of α; and so on.

Listing 4 *Loop_Gen()*: Reverse code generation within loops

Input: An instruction α in a loop L, current traversal count t over L
Output: A RIG for α
begin
1 ζ = Gen_RIG(instance(α, t))
2 RIG_α += ζ
3 **if** ζ uses instructions from within L **then**
4 return RIG_α
5 **end if**
6 **if** $t == 3$ **then**
7 RIG_α = State_save(α)
8 return RIG_α
9 **end if**
end

Since the instructions within L repeat exactly, if a set of instructions generated to reverse the effect of an instance of α makes use of the instructions within L only, then that set can be used to reverse the effect of all the later instances of α as well. In this way, *Loop_Gen()* can decide on when to stop the traversals over L.

Ideally, the traversals over L should be repeated until *Loop_Gen()* can construct a set that makes use of the instructions within L only. However, we limit the maximum number of traversals over a loop body to three not only to limit the time cost of the RCG algorithm but also to limit the length of the reverse code generated for α. This number is arbitrarily chosen and can be increased at the expense of having a larger reverse program. If a set cannot be constructed within three traversals over L, we apply state saving to generate a RIG which reverses the effect of all the instances of α (line 7). In case state saving can be avoided, on the other hand, the generated sets of instructions at each traversal (i.e., the sets from ζ_1 up to ζ_3) are combined together to produce a RIG for α (line 2). The set of instructions to be executed within the RIG during a specific instruction-level reverse execution is determined by the help of a loop counter which distinguishes among different loop iterations.

The following example illustrates reverse code generation for loops.

Example 4.6 Figure 6 shows a symbolic version of the generated RIG for the first instruction α in the loop of Figure 5. Figure 6 also shows the loop unrolled three times where each unrolled iteration corresponds to one of the traversals of the RCG algorithm over the loop body.

At the first traversal of the loop, *Gen_RIG()* finds the reaching definition of the destroyed register r_1 as "$r_1 = 0$" at point **P1**. Then, *Gen_RIG()* generates the set ζ_1 as "$r_1 = 0$" which reverses the effect of the first instance of α (i.e., δ_4) by using the redefine technique.

At the second traversal of the loop, the definition of r_1 to be recovered is the definition that reaches **P2**. This definition is "$\delta_4 : r_1 = r_2 + 2$" which comes from within the loop this time. In order to recover r_1 from δ_4, the RCG algorithm needs the value of r_2. However, r_2 is destroyed by δ_5 between δ_4 and **P2**. The destroyed definition of r_2 is "$\delta_2 : r_2 = 3$" which comes from outside of the loop. Therefore, *Gen_RIG()* first puts into ζ_2 the instruction "$r_t = 3$" which restores the value of r_2 into a temporary register r_t using the redefine technique (r_t is used instead of r_2 to preserve the current value of r_2). Then, the *Gen_RIG()* puts into ζ_2 the instruction "$r_1 = r_t + 2$" which recovers the destroyed value of r_1.

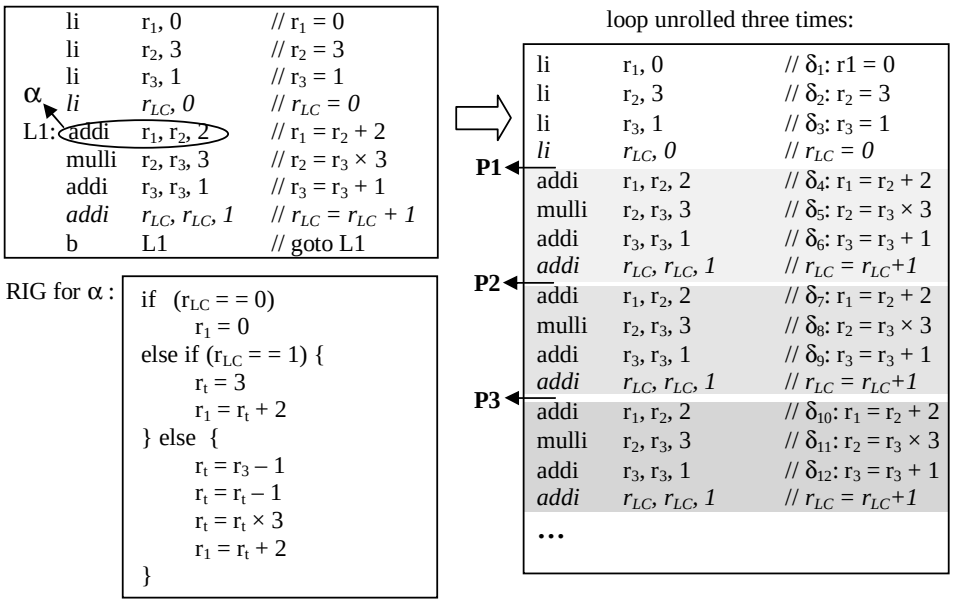


Figure 6: A diagram which illustrates reverse code generation for loops.

At the third traversal of the loop, we are at point **P3**. The reaching definition of r_1 at **P3** is "$\delta_7 : r_1 = r_2 + 2$". The definition of r_2 used in δ_7 is "$\delta_5 : r_2 = r_3 \times 3$" and is destroyed by δ_8 before reaching **P3**. Therefore, we have to recover r_2 before recovering r_1. However, r_3 as used in δ_5 does not reach point **P3**, either. Moreover, r_3 has been overwritten twice after being used in δ_5: once by δ_6 and once by δ_9. Thus, *Gen_RIG()* first puts into ζ_3 the instructions "$r_t = r_3 - 1$" and "$r_t = r_t - 1$" which restore the value of r_3 into a temporary r_t by using the extract-from-use technique twice (once on δ_9 and once on δ_6). Then, *Gen_RIG()* puts into ζ_3 the instruction "$r_t = r_t \times 3$" which restores the value of r_2 into r_t by using the redefine technique. Finally, *Gen_RIG()* puts into ζ_3 the instruction "$r_1 = r_t + 2$" which recovers the value of r_1. Since ζ_3 is constructed using instructions only from within the loop, ζ_3 indeed reverses the effect of the later instances of α as well. Therefore, for this example, it is sufficient to traverse the loop three times to generate a RIG for α without state saving. As seen in Figure 6, the set (ζ_1 or ζ_2 or ζ_3) to be executed within the generated RIG during instruction-level reverse execution is determined by a loop counter (r_{LC}) which is inserted into the original loop.

Note that the generated reverse code in this example is unoptimized. However, the reverse code can be easily optimized by a separate pass using standard optimization techniques such as constant propagation or common subexpression elimination. □

The technique described in this section is applied in a straightforward way to the nested loop structures as well wherein the passes over the nested loops are completed starting from the innermost loop going to the outermost loop.

4.8 *Combine_RIGs()*: Combining the reverse instruction groups

The function *Combine_RIGs()* (called from line 14 of Listing 1) combines a RIG with all the previously generated RIGs to generate an up-to-date, given the RIGs generated so far,

reverse version of the program partition under consideration. The pseudo code for the *Combine_RIGs()* function is given in Listing 5.

Listing 5 *Combine_RIGs()*: Combining the reverse instruction groups

Input: A RIG, RIG_α, generated for an instruction α
Output: A linked list of RIGs
begin
1 **if** α is beginning of a basic block BB_k **then**
2 $n = |IncomingEdges(BB_k)|$
3 **if** $n > 1$ **then**
4 **for** $z = 1$ to $n - 1$ **do**
5 Generate a set C of conditional branch instructions with the predicates determined by *Find_CF()*
6 Link C to the top of the reverse code
7 **end for**
8 **else if** BB_k is a target of a conditional branch instruction β in the original code **then**
9 Generate an unconditional branch instruction ub
10 Link ub to the top of the reverse code
11 **end if**
12 **end if**
13 Link RIG_α to the top of the reverse code
end

As we mentioned in the beginning of Section 4, the generated RIGs should be placed in a way to make the RIGs execute in an order opposite to the completion order of the instructions in the original program partition. We know that the instructions within a BB complete in lexical order; therefore, placing the RIGs in the order opposite to the lexical order of the original BB is sufficient to generate the reverse of that BB. This implies that the RIGs generated for the BBs are placed in a *bottom-up* fashion in the reverse code (line 13 of Listing 5). In other words, if a basic block BB_i in the program partition under consideration has a sequence of instructions $I_{BB_i} = (\alpha_1, \alpha_2, \alpha_3, \ldots \alpha_n)$, and if the corresponding RIGs generated for BB_i are $RIG_{BB_i} = \{RIG_1, RIG_2, RIG_3, \ldots RIG_n\}$, then the reverse of BB_i, designated as RBB_i, consists of the sequence $I_{RBB_i} = (RIG_n, RIG_{n-1}, RIG_{n-2}, \ldots RIG_1)$. Note that since a generated RIG, RIG_k $(1 \leq k \leq n)$ in I_{RBB_i}, may contain branch instructions (see Example 4.4), an RBB may not necessarily be a single basic block, but instead may be a combination of multiple basic blocks. The following example shows how the RBBs are constructed from the BBs of a program partition.

Example 4.7 *Construction of the RBBs*: Figures 7(a) and 7(b) show the PCFG of foo and the RBBs generated for $reverse\_foo$, respectively. The RCG algorithm generates the reverse of each BB in foo by combining the generated RIGs in bottom-up placement order in $reverse\_foo$. While the reverse of BB1, BB2 and BB3 (namely, RBB1, RBB2 and RBB3) are constructed each as a single BB, the reverse of BB4, RBB4, consists of three separate BBs. RBB4 is separated into three BBs because the reverse of the instruction "$r_{12} = r_{11} \times r_{10}$" in BB4 consists of multiple instructions two of which are branches. Note that since the initial values of r_{10}, r_{11} and r_{12} are input to foo (and thus the redefine technique is not applicable) and since these initial values are not used in foo (and thus the extract-from-use technique is not applicable either), the initial values of r_{10}, r_{11} and r_{12} are recovered

in $RBB1$ by the state saving method described in Section 4.6. Figure 7(c) shows the PCFG of foo instrumented with the state saving instructions. □

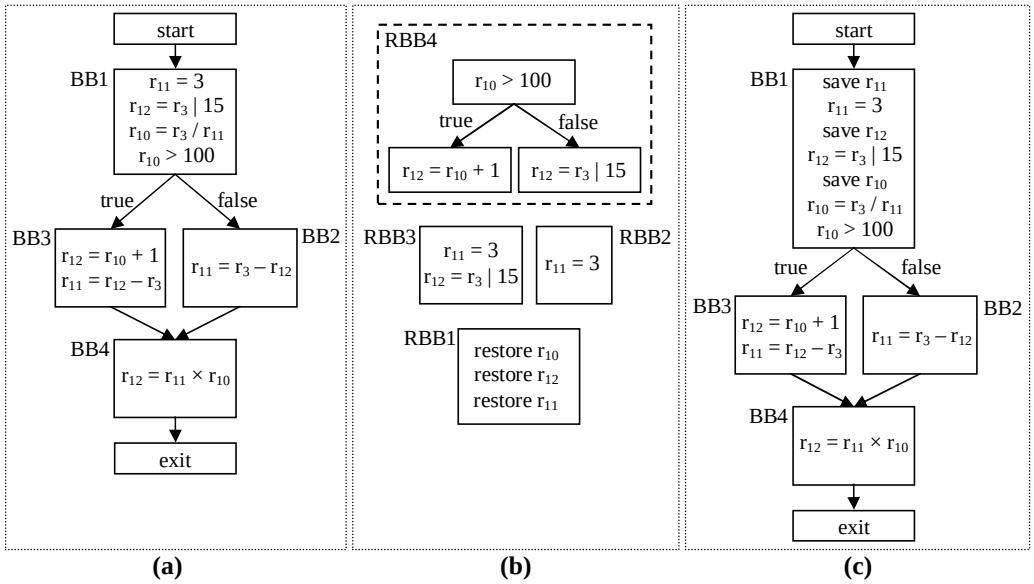


Figure 7: (a) PCFG of foo. (b) RBBs of $reverse\_foo$. (c) PCFG of instrumented foo.

4.8.1 Constructing the Reverse Program Partitions

To generate the reverse version of a program partition, the RBBs generated for that program partition should be combined together in an appropriate way. Once again, this combination should satisfy our argument that the RIGs should execute in the order opposite to the completion order of the instructions in the original program. *Combine_RIGs()* achieves this by combining the RBBs via the inverted versions of the edges in the original program partition. Consequently, a confluence point of incoming edges in a program partition typically becomes a fork point of outgoing edges in the reverse version of that program partition, and vice versa.

Suppose that a confluence point P_o in the original program partition becomes a fork point F_r in the reverse program partition. Depending on the number of incoming edges to P_o (or outgoing edges from F_r), *Combine_RIGs()* inserts at F_r one or more conditional branch instructions which determine which edge to take at F_r during reverse execution. As described in Section 4.2, *Find_CF()* associates with each incoming edge of P_o a predicate expression which determines along which edge P_o is reached. Consequently, the predicate expressions associated with the incoming edges of P_o are directly used as the predicates of the conditional branch instructions inserted at F_r. The values of these predicates can either be saved or reevaluated as explained in Section 4.2.

Recall from Section 4.2 that since the predicate expressions found at P_o are mutually exclusive, it is sufficient to save or reevaluate only $n-1$ of the n predicate expressions associated with n incoming edges of P_o. Therefore, at the corresponding fork point F_r in the reverse program partition, *Combine_RIGs()* generates $n-1$ conditional branch instructions, each using one of the $n-1$ predicate expressions found at P_o (lines 3 to 6 of Listing 5).

Due to linear orientation of code in memory, the target of one of the n outgoing edges from F_r immediately follows F_r in address space. Let us name this outgoing edge as e. Since it is inefficient to generate a conditional branch whose target address is the next address, it is not appropriate to generate a conditional branch corresponding to e. Therefore, among the n predicate expressions found at P_o, the predicate expression we leave out is always the one that is associated with the incoming edge of P_o whose inverted version is e (see Example 4.8).

On the other hand, suppose that a fork point F_o of two edges in the original program partition becomes a confluence point P_r of two edges in the reverse program partition (recall from Section 4.1 that a fork point can have at most two outgoing edges at the assembly level). In this case, it is necessary to establish a link between the source of each joining edge and the confluence point P_r in the reverse program partition. Note that as the RCG algorithm analyzes the instructions on the fall-through path of F_o before the instructions on the target path of F_o (due to lexical order scanning of the instructions), the reverses of the instructions on the fall-through path are generated before the reverses of the instructions on the target path. To keep the bottom-up placement order, *Combine_RIGs()* places the reverses of the instructions on the fall-through path below the reverses of the instructions on the target path. Thus, the reverses of the instructions on the fall-through path of F_o always precede P_r in the reverse program partition, which establishes an automatic link between them. Therefore, the remaining part is to provide the link between P_r and the source of the joining edge that is the inverted version of the edge on the target path of F_o. This link is established by inserting an unconditional branch at the source of this joining edge in the reverse program partition (lines 8 to 10 of Listing 5).

The following example illustrates how the RBBs are combined to generate a reverse version of a program partition.

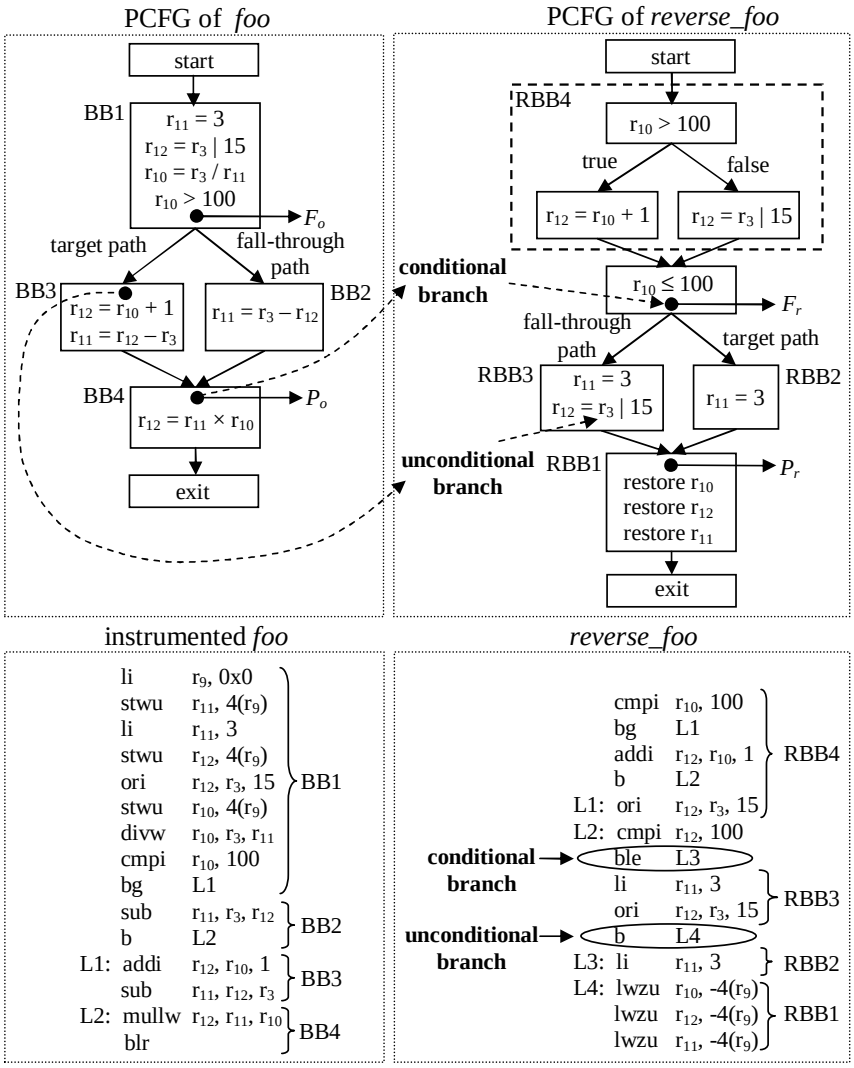
Example 4.8 *Combining the RBBs*: Figure 8 shows the PCFGs of foo and $reverse\_foo$ together. Also seen in the figure are the assembly listings of the instrumented foo (i.e., instrumented with state saving instructions) and $reverse\_foo$. Since the RBBs are combined with the inverted versions of the edges in the PCFG of foo, the confluence point designated as P_o in the PCFG of foo becomes a fork point designated as F_r in the PCFG of $reverse\_foo$, and the fork point designated as F_o in the PCFG of foo becomes a confluence point designated as P_r in the PCFG of $reverse\_foo$. Consequently, a conditional branch instruction is inserted at point F_r (lines 3 to 6 of Listing 5) and an unconditional branch instruction is inserted at the head of one of the joining edges at P_r (lines 8 to 10 of Listing 5). The predicate of the conditional branch inserted at F_r can be determined by observing the following facts:

(1) As already shown in Example 4.1, the predicate expression associated with the left incoming edge of P_o is $r_{10} > 100$ and with the right incoming edge of P_o is $r_{10} \leq 100$. Consequently, control should be directed from point F_r to RBB3 if $r_{10} > 100$ is true and to RBB2 if $r_{10} \leq 100$ is true.

(2) Since the predicate expressions in (1) are mutually exclusive (i.e., they cannot be true at the same time), using only one of the predicate expressions is sufficient to determine the dynamically taken edge to P_o (and thus the edge to be taken out of F_r).

(3) Note that due to the bottom-up placement order, RBB2 is placed below RBB3; therefore, RBB3 follows point F_r in address space.

(4) We know that a conditional branch instruction directs the control to its target address if the predicate of the conditional branch is true; otherwise, execution continues with the instruction after the conditional branch.



The PowerPC 860 instructions "stwu" and "lwzu" are used as push-like and pop-like instructions with r_9 being used as a memory pointer for state saving

Figure 8: A diagram which illustrates the combination of the RBBs.

Therefore, from (1) to (4), the predicate of the conditional branch at F_r is determined as $r_{10} \leq 100$. The value of this predicate expression is dynamically determined by executing a compare instruction "$cmpi\ r_{10},\ 100$" in $reverse\_foo$ (i.e., by reevaluating the predicate value during reverse execution).

Note that due to the bottom-up placement order described, an unconditional branch instruction is placed only at the point that corresponds to the target address of the conditional branch instruction in foo (the other edge simply falls through, i.e., RBB2 is directly followed in address space by RBB1). □

4.9 *Combine_Partitions()*: Combining the reverse program partitions

After generating the reverse version of a program partition, *Combine_Partitions()* (called from line 27 of Listing 1) combines the reverse version of that program partition with the other reverse program partitions that have already been generated. In order to achieve

this, *Combine_Partitions()* must know the control flow information between the program partitions.

4.9.1 *Grow_CG()*: Determining inter-partitional control flow

Since the PCFG construction is performed for each program partition separately, each PCFG designates the control flow within a particular program partition only. In other words, a PCFG does not contain any edges which show the flow of control between the program partitions. Therefore, the control flow information between the program partitions is determined by a *call graph*, CG=(N,E, s, t), which is built by the function *Grow_CG()* mentioned in Section 4.1. The set N of CG is the set of nodes designating the program partitions in the program and the set E of CG is the set of edges designating the flow of control between those partitions. The notations s and t designate the unique entry and exit nodes of the CG. Note that since an indirect call (i.e., use of a function pointer) whose target program partition is statically unknown may potentially invoke any high-level (or unpartitioned) procedure/function in the program under consideration (we assume a function pointer can only point to the beginning address of a high-level procedure/function but not to an arbitrary address), *Grow_CG()* inserts an edge from a program partition F to every other program partition that F may call if F makes an indirect call whose target program partition is statically unknown. To learn from which address(es) a program partition can be immediately reached and thus to be able to move the control backwards to a source address, *Grow_CG()* annotates an edge $e_{ij} \in E$ from a program partition F_i to another program partition F_j with the address of the instruction in F_i that transfers control from F_i to F_j.

Listing 6 shows the pseudo code of *Grow_CG()*. *Grow_CG()* adds a new node to the CG for a program partition when a PCFG is built for that program partition (see line 14 of Listing 2). After a new node n_j is generated for a program partition F_j, *Grow_CG()* checks the program partitions which are immediately reachable from F_j. For every program partition F_k which is immediately reachable from F_j and for which a PCFG (and thus a node in the CG) is generated, *Grow_CG()* adds an edge e_{jk} from the node of F_j to the node of F_k and annotates e_{jk} with the address of the instruction transferring control from F_j to F_k (lines 2 to 5 of Listing 6). For every other program partition which is immediately reachable from F_j but for which a node has not yet been generated, *Grow_CG()* sets a pending edge (lines 6 and 7 of Listing 6). Then, *Grow_CG()* checks whether F_j has pending incoming edges set for it. If F_j has pending incoming edges, *Grow_CG()* adds to the CG all the pending incoming edges that are set for F_j and annotates those edges appropriately (lines 10 to 15 of Listing 6).

Example 4.9 *Call graph construction*: An example program and the corresponding CG are shown in Figure 9. The example program consists of three high-level functions $main$, g and h where $main$ calls g and g makes an indirect call to a high-level function which is not statically known. Since $main$ and g both contain call instructions, *Init_RCG* partitions $main$ into two parts (m_1 and m_2), and g into two parts (g_1 and g_2) by constructing the corresponding PCFGs (see Listing 2). The edges in the CG show the transfer of control between the program partitions and are annotated with the addresses of the instructions transferring control between the partitions. Note that the indirect function call in g_1 may potentially call g_1 itself, m_1 and h but not g_2 nor m_2 because the beginning addresses of g_2 and m_2 do not correspond to the beginning address of any high-level function in the original program. □

Listing 6 *Grow_CG()*: The CG construction algorithm

Input: A program partition F_j for which a PCFG has been generated
Output: A node n_j in the CG with a set of edges connected to n_j
begin

```
 1 Add a node n_j to CG for F_j
 2 for all F_k immediately reached from F_j do
 3   if (n_k = node_of(F_k)) ≠ NULL then
 4     Add to the CG an edge e_jk from n_j to n_k
 5     Annotate e_jk
 6   else
 7     Set e_jk as pending
 8   end if
 9 end for
10 if n_j has a pending incoming edge then
11   for all e_ij from n_i to n_j do
12     Add to the CG an edge e_ij from n_i to n_j
13     Annotate e_ij
14   end for
15 end if
end
```

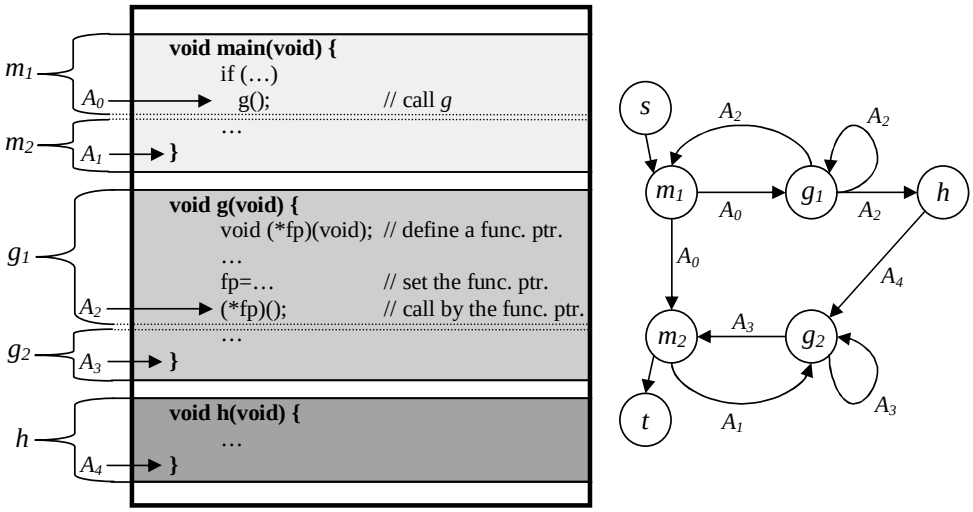


Figure 9: An example call graph (CG).

4.9.2 Using the CG to combine the reverse program partitions

Combine_Partitions() combines the reverse program partitions by systematically inverting the edges in the CG of the original program T when generating the reverse program RT. Consequently, in RT, *Combine_Partitions()* inserts branch or jump instructions at those points which correspond to the destination locations of the edges in the CG of T.

Listing 7 shows the pseudo code for *Combine_Partitions()*. *Combine_Partitions()* consists of five parts (namely, a, b, c, d and e). Parts a to c are related to the program partitions that are immediately reached from multiple static locations in T. Parts d and e, on the other hand, are related to the program partitions that are immediately reached from a single static location each. If a program partition F is immediately reached from a single static

location whose address is A in the program under consideration (i.e., there is a single edge coming to the node of F in the CG and that edge has an annotation A), then in the reverse code, the address RA corresponding to A is the unique address to which the control has to be directed after the reverse of F, RF, is executed. This is easily handled by inserting at the end of RF an unconditional branch instruction whose target address is RA (line e.1 of Listing 7). However, if a program partition F is immediately reached from multiple static locations (i.e., there are multiple edges coming into the node of F in the CG), the location from which F is immediately reached during a specific execution of the program and thus the corresponding location in the reverse code to which the control should be directed after executing RF cannot be obtained statically. Therefore, in such a case, *Combine_Partitions()* applies a dynamic technique, called the *stack-tracing technique*, to find the location to which the control should be directed after executing RF. We will describe the stack-tracing technique in the following paragraphs.

The stack-tracing technique

The stack-tracing technique can simply be described as saving the statically unknown return addresses of reverse partitions into a stack at runtime. During reverse execution, the saved addresses are popped back from the stack to provide return from a reverse partition.

Let us assume that a subset Φ_F of the program partitions in the program under consideration are immediately reached from multiple static locations. We will designate the set of the reverses of these program partitions as Φ_{RF}. Thus, the remaining partitions in the program are immediately reached from a single static location each. Also, assume that there are a total of n locations from which control reaches the program partitions in Φ_F. We will designate the addresses of these n locations as $\Phi_A = \{A_0, A_1, \ldots A_{n-1}\}$ where a subscript shows the unique *array index* associated with a particular address. We will also designate the corresponding n addresses in the reverse code as $\Phi_{RA} = \{RA_0, RA_1, \ldots RA_{n-1}\}$. Therefore, after executing the reverse of a program partition $F \in \Phi_F$ during instruction-level reverse execution, control should be directed to an address RA_{i_d} if and only if the control has reached F from the corresponding address A_{i_d} during forward execution ($A_{i_d} \in \Phi_A$, $RA_{i_d} \in \Phi_{RA}$).

The addresses to which the control should be transferred from a reverse program partition in Φ_{RF} during a specific reverse execution of the program under consideration can be found by saving the addresses in Φ_{RA} into a runtime stack during forward execution. In other words, whenever a transfer from an address A_i in Φ_A to a partition F_j in Φ_F occurs during forward execution, one can save the corresponding reverse address RA_i in Φ_{RA} to provide a return from the reverse of F_j to address RA_i during reverse execution. However, in a typical processor with a 32-bit address bus (e.g., PowerPC 860), each address is 32-bits in length. In other words, a total of 2^{32} different addresses can be accessed through the address bus. On the other hand, in a typical program, the total number of addresses in Φ_{RA} and thus the maximum array index in Φ_{RA} is typically much less than 2^{32}. Therefore, instead of saving the addresses themselves, the stack-tracing technique saves the array indices of those addresses and then matches the saved indices to the addresses in Φ_{RA}. In this way, the memory requirement for keeping track of the addresses may be reduced.

To possibly reduce the memory requirement further, the indices that are consecutively encountered during program execution and that have the same value (which may happen

Listing 7 *Combine_Partitions()*: Combining the reverse program partitions

a. *At each address $A_j \in \Phi_A$ where a recursive call is made or might be made (we say "might be made" as A_j might be the address of an indirect call site), insert the instructions which perform the following:*

```
1  check the top entry in M
2  if the flag of the top entry is '0' then
3     if the index of the top entry is j then
4        push a new entry with a flag '1' and a counter '2' over the top entry in M
5     else
6        push a new entry with a flag '0' and an index j over the top entry in M
7     end if
8  else /*the flag of the top entry is '1'*/
9     check the entry below the top entry as well
10    if the index of the entry below the top entry is j then
11       increment the counter of the top entry by one
12    else
13       push a new entry with a flag '0' and an index j over the top entry in M
14    end if
15 end if
```

b. *At any other address $A_j \in \Phi_A$, insert the instructions which perform the following:*

```
1  push over the top entry in M an entry which has a flag '0' and an index j
```

c. *At the end of each program partition $RF \in \Phi_{RF}$, insert the instructions which perform the following:*

```
1  check the top entry in M
2  if the flag of the top entry is '0' then
3     extract the index i_d of the top entry and pop the top entry from M
4  else
5     decrement the counter of the top entry by one
6     extract the index i_d of the entry below the top entry
7     if the counter has reached zero then
8        pop the top two entries from M
9     end if
10 end if
11 find in X the address RA_{i_d} ∈ Φ_RA at the offset i_d × |A| from B and branch to RA_{i_d}
```

d. *At the end of each reverse program partition $RF \notin \Phi_{RF}$ of which forward program partition F is called indirectly from a unique address $A_k \in \Phi_A$, insert the instructions which perform the following:*

```
1  check the top entry in M
2  if the flag of the top entry is '0' then
3     if the index of the top entry is k then
4        pop the top entry from M
5     end if
6  else /*the flag of the top entry is '1'*/
7     check the entry below the top entry as well
8     if the index of the entry below the top entry is k then
9        decrement the counter of the top entry
10       if the counter has reached zero then
11          pop the top two entries from M
12       end if
13    end if
14 end if
15 execute an unconditional branch to the corresponding address RA_k ∈ Φ_RA in the reverse code
```

e. *At the end of each reverse program partition $RF \notin \Phi_{RF}$ of which forward program partition F is immediately reached from an address $A \notin \Phi_A$, insert the instructions which perform the following:*

```
1  execute an unconditional branch to the corresponding address RA ∉ Φ_RA in the reverse code
```

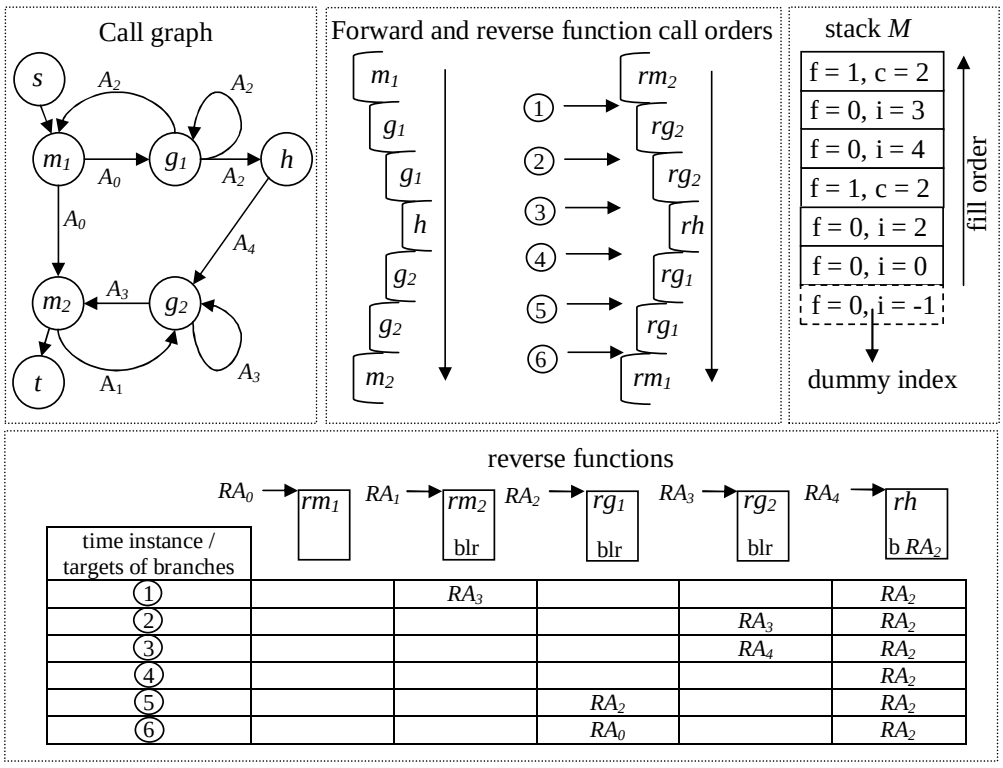
with recursive function calls) are compressed into two memory locations by the stack-tracing technique. The first memory location holds the repeating index value and the second memory location holds a counter which shows how many times the index value repeats. Thus, the stack-tracing technique keeps the following two data structures to keep track of the addresses from which the control is transferred to the program partitions in Φ_F.

The first data structure is to keep the associated array indices of the addresses from which control *dynamically* reaches the program partitions in Φ_F during a specific execution of the program. For this purpose, the stack-tracing technique uses a memory space M which is accessed like stack. We choose the length of an entry in M to be sixteen bits. Each entry is a 1-bit flag concatenated with a 15-bit value. Depending on the value of the flag bit, the 15-bit value designates either the array index j of an address $A_j \in \Phi_A$ (if the flag bit is '0') or a counter value (mentioned in the previous paragraph) which tells how many consecutive index values the next entry in M represents (if the flag bit is '1'). When an entry is to be made into M, the last entry in M is checked for a possible compression opportunity of the new entry (see line a.1 of Listing 7). In order to prevent an accidental compression of the first entry into M, which may happen if the irrelevant memory value just before M indicates a valid index which is same as the first entry into M, the stack-tracing technique initially inserts into M an entry with a flag '0' and a dummy index which cannot be equal to the index of any address in Φ_A (or Φ_{RA}).

The second data structure is to keep all the addresses in Φ_{RA} to one of which the control can immediately be directed after executing a reverse program partition in Φ_{RF}. Hence, the indices recorded in M can be matched to the addresses kept in this second data structure. For this purpose, the stack-tracing technique uses an array X in which all the addresses RA_0 to RA_{n-1} are consecutively stored starting from a base address, say, B. Therefore, an address RA_j $(0 \leq j \leq n-1)$ is placed at a byte offset $j \times |A|$ from B where $|A|$ is the length (in bytes) of an address on the target processor and j is the index corresponding to RA_j (and thus to A_j). Note that obviously, X is constructed after the reverse program is generated and the addresses in Φ_{RA} are resolved.

Then, the stack-tracing technique inserts instructions both into the original and the reverse code to invert the control flow between the program partitions that are immediately reached from multiple static locations. The instructions inserted into the original code handle the bookkeeping task by saving the dynamically encountered indices into M and apply the mentioned compression for repeating index values (parts a and b of Listing 7). The instructions inserted into the reverse code, on the other hand, retrieve the saved indices from M, match the retrieved indices to the addresses stored in X and transfer the control to the dynamically found addresses in this way (parts c and d of Listing 7).

Example 4.10 *Combining the reverse versions of the program partitions*: Figure 10 illustrates how *Combine_Partitions()* combines the reverse versions of the program partitions. A sample function call history and the corresponding reverse function call order of the example program given in Figure 9 are shown in Figure 10. According to the sample function call history, the program under consideration is forward executed starting from the beginning of m_1 until the end of m_2. Then, this execution is reversed by executing the corresponding reverse program starting from the beginning of rm_2 (the reverse of m_2) until the end of rm_1 (the reverse of m_1). The reverse function call sequence is marked with timestamps which indicate the instances when the control is transferred between the reverse functions.



f: flag, c: counter, i: index, blr: branch to link register, b: unconditional branch

Figure 10: An example of combining the reverse program partitions.

Since functions m_2, g_1 and g_2 can be immediately reached from multiple static locations, *Combine_Partitions()* inserts indirect branch (branch to link register) instructions to the end of the corresponding reverse functions rm_2, rg_1, rg_2 where the target addresses of these indirect branch instructions are determined dynamically. On the other hand, since function h is immediately reached from a single static location, an unconditional branch instruction with the hard-coded target address RA_2 of the unique call location of h (the end of g_1) is inserted to the end of the corresponding reverse function rh (line e.1 of Listing 7). Figure 10 also shows a table which indicates the dynamically and statically determined addresses of the branch instructions and at what timestamp instances those addresses are determined.

The final state of the data structure M at the end of forward execution of the program is shown in Figure 10. Initially, M contains a dummy index entry to prevent the compression of the first valid entry into M. At the end of function m_1, when a call is made to function g_1, the index '0' of the address A_0 (which is the address of the call location) is entered into M with a flag of '0' which indicates that the entry is an index value (line a.6). Then, at the end of function g_1, a recursive call is made to g_1, and the index '2' of the address A_2 is entered with a flag '0' over the previous entry in M (line a.6). When the call point at the end of g_1 is reached again, the index '2' is supposed to be entered into M again; however, since the index '2' repeats, a counter '2' with a flag '1' is entered into M instead (line a.4). Similar steps are followed to enter the rest of the indices into M.

During reverse execution, the stack-tracing technique determines the target addresses of the indirect branch instructions at the end of the reverse functions by checking the entries in M. At the end of the reverse function rm_2, the top entry in M is checked (line c.1). Since the top entry represents a counter

value, the counter value '2' is decremented by one (line c.5) and the index '3' of the entry below the top entry is extracted (line c.6). Then, the address RA_3 corresponding to the extracted index '3' is found in X and an indirect branch is executed to the found address RA_3 which is the beginning address of rg_2 (line c.11). When the end of rg_2 is reached during reverse execution, the top entry in M is checked (line c.1). Since the top entry is again a counter value, the counter value, which is now '1', is decremented by one (line c.5) and the index '3' of the entry below the top entry is extracted (line c.5). However, since the counter has reached '0', the top two entries in M are popped this time (line c.8). Then, the address RA_3 corresponding to the extracted index '3' is found in X and an indirect branch is executed to the found address RA_3. Similar steps are followed during the rest of the reverse execution, which results in the correct ordering of the reverse function calls shown in Figure 10. □

4.10 Summary of the overall RCG algorithm

The overall RCG algorithm is summarized in the flowchart in Figure 11. The RCG algorithm first constructs a PCFG with labeled edges for every program partition in a program and constructs the CG of the program (Box 1 in Figure 11). Then, the RCG algorithm enters a main loop where the instructions of each program partition are read one after another and the reverse program is built. At a confluence point of two or more edges in the PCFG of the program partition currently being analyzed, the algorithm finds the predicate expressions which control via which incoming edge the confluence point will be reached dynamically (Group 1 in Figure 11).

After an instruction is read, the RCG algorithm checks whether the instruction directly modifies a register or a memory value. If yes, the RCG algorithm generates a RIG for the read instruction. If the instruction is outside a loop, the RCG algorithm generates the RIG by calling *Gen_RIG()* directly (Box2); otherwise, the RCG algorithm calls *Loop_Gen()* to generate a RIG for the instruction (Group2).

After a RIG is generated for an analyzed instruction, that RIG is written into the reverse program partition that is currently being constructed (Box 3). As described in Section 4.7.1, some instructions within a loop require more than one pass over the loop body (excluding the initial pass over the whole program to generate the PCFGs and the CG) before reverse code can be generated for those instructions without state saving. Therefore, if an analyzed instruction is inside a loop and the generation of a RIG which does not use any state saving for reversing the analyzed instruction requires another pass over the loop body, the RCG algorithm traverses the loop body once more provided that the total number of passes over the loop body will not exceed three.

When the construction of the current reverse program partition is completed, the RCG algorithm connects the constructed reverse program partition to the rest of the reverse program (Box 4).

5 Filling in the Details of the RCG Algorithm

In this section, we present the detailed descriptions of the PCFG edge-labeling algorithm, predicate expression determination and the generation of the RIGs which have been omitted in the overview of the RCG algorithm in Section 4. If a detailed understanding of the RCG algorithm is not required, the reader may skip this section and move directly to Section 6.

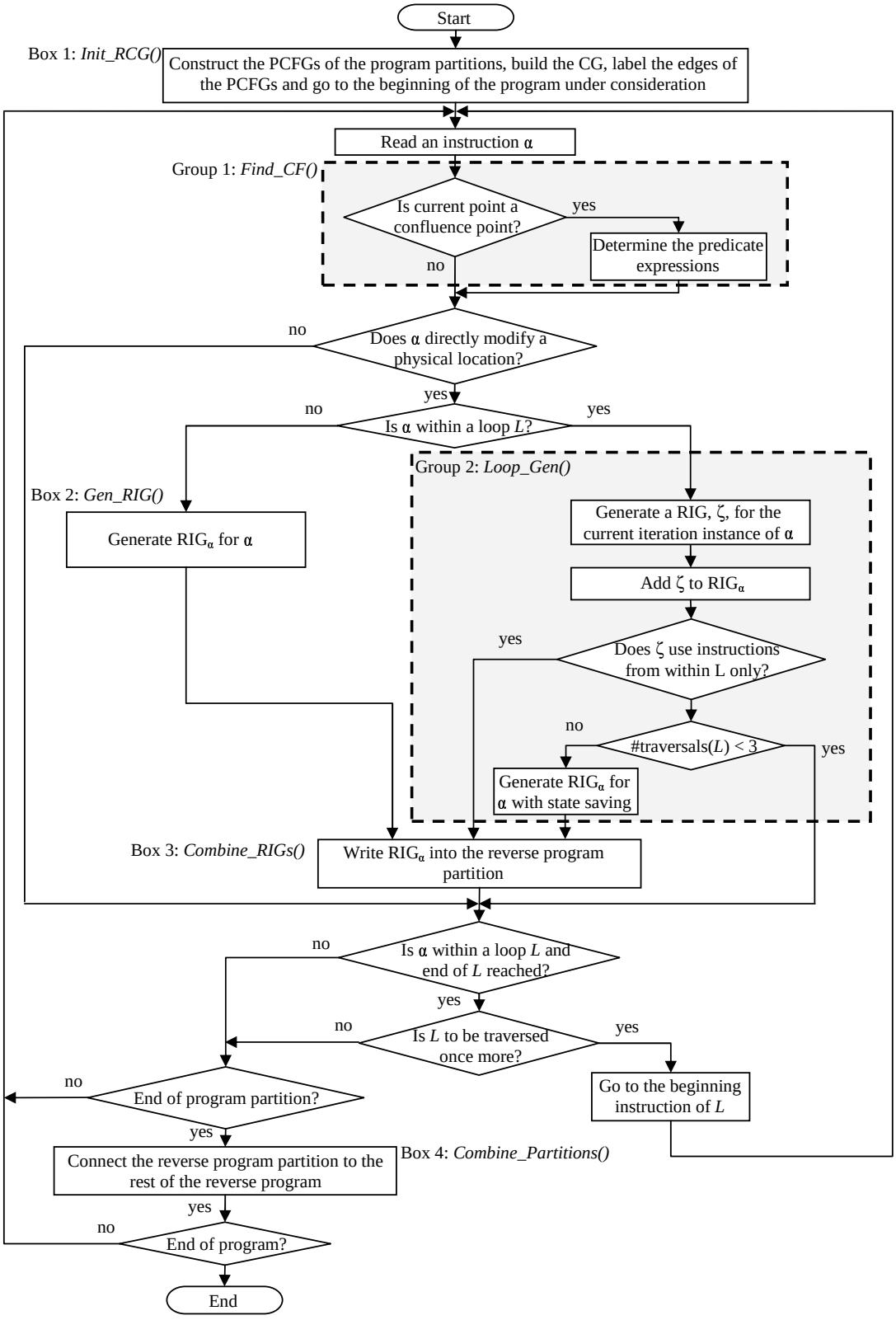


Figure 11: A high-level flowchart of the RCG algorithm.

5.1 PCFG edge-labeling algorithm

As mentioned before, PCFG labeling is performed for determining control flow predicates and reaching definitions in an efficient way at a particular program point. Edge labeling is performed by the function *Label_edges()* which is called by *Init_RCG()* on lines 5 and 10 of Listing 2. *Label_edges()* assigns a special label to every forward edge in the PCFG of a program partition. Backward edges are not considered because giving labels to backward edges helps neither in the determination of the predicate expressions nor reaching definitions.

Each label assigned to an edge indicates the union of one or more closed intervals on a bounded nonnegative integer number axis. We name an interval [x,y] as a *control flow interval* (*CFI*) and assign the interval [x,y] 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 PCFG 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 PCFG. 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. To better understand the control flow regions, consider the following example.

Example 5.1 *Control flow regions*: Figure 12 shows an example PCFG in which the control flow regions are marked. In the figure, 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 12, 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 PCFG of a program partition 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.

We choose 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 program partition 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 can hardly ever be seen in a program partition. Therefore, for all practical purposes, bounding the integer number axis between zero and $2^{32} - 1$ will be more than enough for *Label_edges()* to function correctly. The code for handling greater than 31 nested conditionals is a special case which will rarely, if ever, be invoked.

Listing 8 shows the operations *Label_edges()* performs on the edges of the BBs in a PCFG. In Listing 8, the notation $L^{in}_{i,j}$ ($L^{out}_{i,j}$) designates the label of the j^{th} incoming (outgoing) forward edge $\in$ InFwdEdges ($\in$ OutFwdEdges) of a basic block BB_i. Please note that a label $L^{in}_{i,j}$ or $L^{out}_{i,j}$ consists of a set of one or more intervals or CFIs. *Label_edges()* 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 (line 2 of Listing 8). If BB_i is not the **start** block, *Label_edges()*

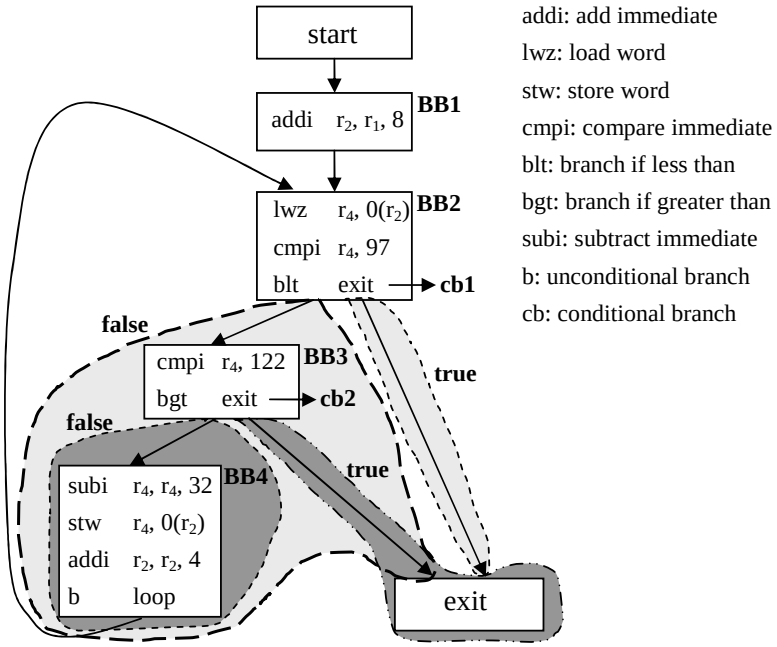


Figure 12: An example PCFG that shows the control flow regions.

first calculates the union of the labels of the incoming forward edges of BB_i where the union operation is performed on the intervals indicated by the labels of the incoming forward edges (line 4). After the union operation, if BB_i has two outgoing forward edges, *Label_edges()* divides each interval designated by the union of the incoming forward edge labels into two equal portions. Then, *Label_edges()* assigns the union of the lower portions (coming from each interval) as a label $L_{i,1}^{out}$ to the outgoing forward edge on the fall-through path (lines 5, 6 and 7). The union of the upper portions, on the other hand, is assigned as a label $L_{i,2}^{out}$ to the outgoing forward edge on the target path (lines 5, 6 and 8). If BB_i has only one outgoing forward edge, *Label_edges()* assigns the union of the incoming forward edge labels to that edge without any change (lines 5, 9 and 10). The following example illustrates the edge-labeling algorithm.

Example 5.2 *Edge-labeling algorithm*: Figure 13 shows the PCFG of Figure 12 with its edges labeled. For this example, the parameter t shown in Listing 8 is chosen as 8. Therefore, the outgoing edge of the **start** block 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. BB3 has two outgoing forward edges as well. Therefore, *Label_edges()* divides the label [0,127] of the incoming edge of BB3 into two equal portions [0,63] and [64,127] and assigns each portion to one of the outgoing forward edges of BB3. Since BB4 has no outgoing forward edge, no labeling occurs for BB4. All the CFIs formed are shown in Figure 14. Note that in this example, each label consists of a single interval. □

Listing 8 *Label_edges()*: The PCFG edge-labeling algorithm

Input: A basic block BB_i
Output: A Label for each outgoing forward edge of BB_i
begin

```
 1 if BB_i = start block then
 2   L^out_{i,1} = [0,2^t − 1] /*note that t is a global constant; typically, t = 32*/
 3 else
 4   ⋃_{k=1}^{n} [x_k,y_k] = ⋃_{j=1}^{|InFwdEdges(BB_i)|} L^in_{i,j}
 5   for k = 1 to n do
 6     if |OutFwdEdges(BB_i)| = 2 then
 7       L^out_{i,1} ∪= [x_k,(x_k + y_k + 1)/2 - 1]
 8       L^out_{i,2} ∪= [(x_k + y_k + 1)/2,y_k]
 9     else if |OutFwdEdges(BB_i)| = 1 then
10       L^out_{i,1} ∪= [x_k,y_k]
11     end if
12   end for
13 end if
end
```

5.2 Predicate expression determination

Predicate expression determination is performed by the function *Find_CF()* which is called by the main function of the RCG algorithm (see line 6 of Listing 1). We gave an overview of this function in Section 4.2. Now, we will give the details behind the predicate expression determination.

A confluence point P in a PCFG is dynamically reached via an incoming edge e if the innermost control flow region in which e resides is dynamically visited. Therefore, the predicate expression Υ which, when true, causes P to be reached via an incoming edge e will simply be an appropriate combination of the predicates of the relevant conditional branch instructions which cause the innermost control flow region which contains e to be visited. However, a simplification can be made in Υ in certain cases: Suppose that a particular conditional branch instruction, say *cb*, defines two control flow regions R_{true} (that is under the true branch of *cb*) and R_{false} (that is under the false branch of *cb*). Suppose further that R_{true} (or R_{false}) encapsulates the innermost control flow region in which a particular edge e coming to P resides. Therefore, in order for e to be visited passing through *cb* during a specific execution of the program under consideration, the predicate of *cb* must take the true (or false) value. However, if (1) no other edge coming to P is reached through *cb* or (2) if the other edges coming to P that are also reached through *cb* reside only in R_{true} (or R_{false}) as well, then the predicate of *cb* does not play a role in the separation of the condition that causes P to be visited via e from the conditions that cause P to be visited via the other incoming edges. This is because of the following reason: in both cases (1) and (2) above, if P is reached via an incoming edge that is reached through *cb*, then we definitely know that the predicate of *cb* is true (or false); otherwise, we definitely know that *cb* has not been evaluated at all. Therefore, in either case (1) or (2), the predicate of *cb* can be removed from the predicate expressions determined for the incoming edges that are reached through *cb*.

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

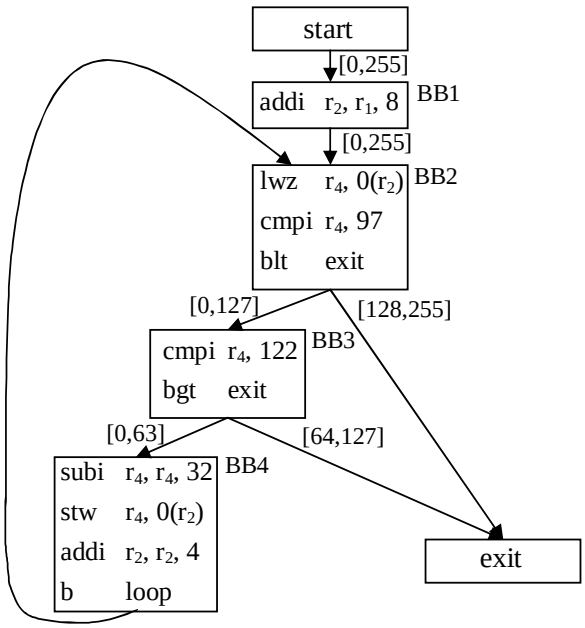


Figure 13: An example PCFG with labeled edges.

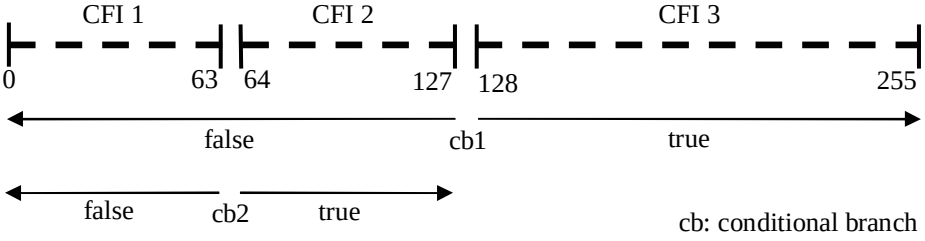


Figure 14: The control flow intervals for the PCFG in Figure 13.

to use can be accomplished very easily by using the edge labels. Consider the following example.

Example 5.3 *Control flow predicate determination*: Suppose that we want to find the predicate expressions which control via which incoming edge the **exit** block in Figure 13 will be reached dynamically. 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 14, [64,127] corresponds to the CFI where the predicate, $r_4 < 97$, of the conditional branch cb1 ("*blt exit*" in Figure 13) is false and the predicate, $r_4 > 122$, of the conditional branch cb2 ("*bgt exit*" in Figure 13) 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 (i.e., a simplification in the predicate expressions can be applied here). Therefore, the predicate expression associated with

the right incoming edge will be $r_4 < 97$, and the predicate expression associated with the left incoming edge will be the complementary predicate expression $r_4 \geq 97$. □

Note that since backward edges are not labeled, the predicate expressions which determine whether a loop will dynamically be reached via an incoming backward edge or a forward edge cannot be found by the method explained. Therefore, in such a case, we follow another approach which we call the *loop counter approach*. Before explaining the loop counter approach, it would be appropriate to make the definitions of some basic terms about a loop as we will use these terms during the explanation of the loop counter approach.

Definition 5.1 *Loop*: In a PCFG, a loop is a *strongly connected component* (*SCC*) of the PCFG. A SCC is a subgraph $G_S = (N_S, E_S)$ of the PCFG, such that there exists a path from every node in N_S to every other node in N_S. A *loop header* $N_{lh} \in N_S$ is a node with an incoming edge from a node which is not in N_S. Since a loop may be entered from multiple points, a loop may have more than one loop header. A *loop preheader* $N_{lp} \notin N_S$ is a node which is an immediate predecessor of a loop header node. A loop is associated with a unique backward edge $e_b \in E_S$ which defines that loop. This backward edge is the outermost backward edge within the loop. A *loop tail* $N_{lt} \in N_S$ is a node which is the source of the backward edge e_b. □

Since a loop can only be entered through a loop header, finding how a loop is reached is equivalent to finding how a loop header is reached. Thus, we only consider loop header blocks. We assume that a loop header block has n incoming backward edges designated as $E_b = \{e_{b_1}, e_{b_2}, \ldots, e_{b_n}\}$ $(n \geq 1)$ and m incoming forward edges designated as $E_f = \{e_{f_1}, e_{f_2}, \ldots, e_{f_m}\}$ $(m \geq 1)$. Since each loop is associated with a unique backward edge, each incoming backward edge in E_b belongs to a different a loop.

Then, the loop counter approach works as follows. *Find_CF()* assigns a dedicated loop counter to each loop defined by each backward edge in E_b. We will designate these loop counters as $LC = \{LC_1, LC_2, LC_3, \ldots, LC_n\}$. At each preheader of each loop, *Find_CF()* inserts an instruction which initializes the corresponding loop counter to zero; and at the loop tail block of each loop, *Find_CF()* inserts an instruction which increments the corresponding loop counter by one. Furthermore, at the reverse version of the loop tail block of each loop, *Find_CF()* inserts an instruction which decrements the corresponding loop counter by one. Therefore, during forward execution, if a loop header block is reached along an incoming forward edge of that block, all the loop counters in LC must have a value of zero; otherwise, at least one of the loop counters in LC must have a value which is greater than zero. Thus, if there is only one forward edge coming to the loop header block, that forward edge is associated with the predicate expression $\Upsilon_f = (LC_1 == 0 \wedge LC_2 == 0 \wedge \ldots \wedge LC_k == 0)$; and if there is more than one forward edge coming to the loop header block, then each predicate expression that is associated with each forward edge (by the explained method in the beginning of this subsection) is ANDed with the predicate expression Υ_f. Each backward edge $e_{b_i} \in E_b$ $(1 \leq i \leq n)$ of the loop header block, on the other hand, is associated with a predicate expression $LC_i > 0$ where LC_i is the loop counter dedicated to the loop containing e_{b_i}.

Note that a loop counter LC associated with a loop L is preferably kept as a register in order to minimize memory and time overheads during forward execution. If a free register

cannot be found to keep LC, an occupied register which is not used within L is freed up by spilling the value in the register into memory at each preheader of L (i.e., just before L is entered). Then, at the beginning of each BB to which there is an exit from L, the spilled value is written back to the register used as LC. However, if a suitable occupied register cannot be found, LC is kept in memory. We illustrate the loop counter approach in Example 5.7 in Section 5.4.

5.3 Details of reverse instruction group generation

In this section, we will give a detailed description of how, given an instruction α, *Gen_RIG()* generates a RIG able to reverse the effects of instruction α.

We mentioned in Section 4.3 that in order to recover the value of a variable destroyed by an instruction α, the first thing to do is to find out the reaching definitions for the variable at the program point just before α (line 4 of Listing 3). This is because the definition destroyed by α is indeed equal to one of the reaching definitions at a specific execution of the program under consideration. We also mentioned in Section 4.3 that *Gen_RIG()* applies a technique called value renaming to find the reaching definitions easily. Therefore, let us now explain the value renaming operation of *Gen_RIG()* and then explain how reaching definitions are determined by *Gen_RIG()*.

5.3.1 Value renaming

Value renaming is the assignment of a different name to every definition of a variable (i.e., a directly modified register or memory location). By value renaming, *Gen_RIG()* can easily distinguish different definitions reaching a particular point in a PCFG.

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

Example 5.4 *Value renaming for registers*: Consider the following instruction sequence:

addi r_2, r_1, *8* $\quad$ $//r_2 = r_1 + 8$
addi r_2, r_2, *4* $\quad$ $//r_2 = r_2 + 4$

The initial values of the registers are given the names r_1^0 and r_2^0 for r_1 and r_2, respectively. Then, the first instruction generates a new value designated by r_2^1 by using the values r_1^0 and '8'. After that, the second instruction generates another value designated by r_2^2 using the values r_2^1 and '4'. □

Renaming memory values is not as easy as renaming register values. This is because a memory location being written by an instruction is not always apparent within the instruction encoding, which is the case for indirect addressing (please note that even if a memory location being written by an instruction is not apparent within the encoding of the instruction, that memory is still directly modified by the instruction if the instruction encoding

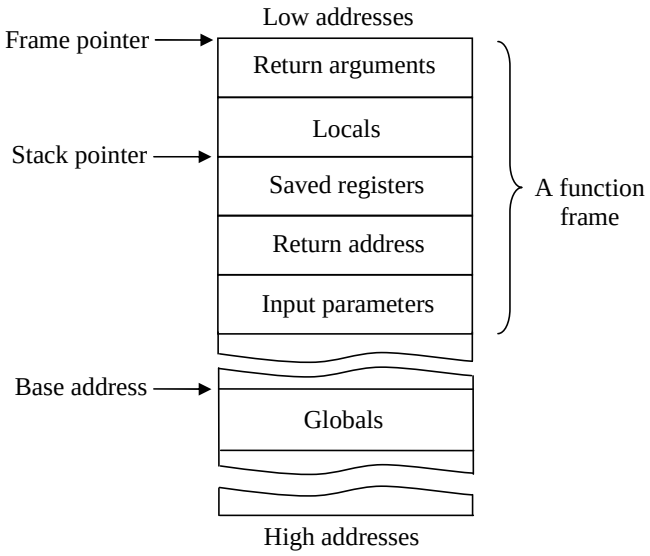


Figure 15: A typical memory organization made by a compiler.

includes at least one operand which is used to point to the modified memory location). Consequently, it might be hard to determine whether two memory stores made by two different instructions are to a same location or not. Fortunately, there is a way to distinguish the target memory location of an unambiguous memory store from the other stores even if the written location is not apparent within the instruction encoding.

Figure 15 shows a memory organization made by a typical compiler. The addresses of all local stores within a program partition can be expressed as a summation of the value of the frame pointer (or the stack pointer if the frame pointer is not available as a dedicated register) and the offset used for the store. The addresses of global stores in a program can be expressed in a similar way, but by using the base address of the global data section of the executable code in place of the frame pointer [3]. The important point here is that the base address of the global data section is fixed throughout the execution of a program and the value of the frame pointer is fixed throughout the execution of a program partition. Therefore, knowing the offset value used for a memory store is sufficient for distinguishing the target location of that memory store from the target locations of the other memory stores in the intra-procedural analysis of the RCG algorithm.

The offset values used for unambiguous memory stores (e.g., those for ordinary variables, pointers with statically known targets and arrays with statically known indices) are statically apparent in an executable code. This means that the locations of unambiguous memory stores can be determined statically and, thus, value renaming can be done for those memory stores without any problem. However, the offset values of ambiguous memory stores (e.g., those for pointers aliased to statically unknown variables or arrays with statically unknown indices) are not statically apparent. If an offset value in a memory store cannot be found statically, we still assign a distinct name to the stored value as if that value were written into a physical memory location that had never been accessed before; however, to be conservative, we assume that the memory store is capable of changing the value of any memory location. The following example illustrates how value renaming is performed for memory locations.

Example 5.5 *Value renaming for memory locations*: Consider the following instruction sequence:

$stw\ \ r_2,\ 4(r_4)$ $\quad$ $//\text{mem}[r_4 + 4] = r_2$
$stw\ \ r_5,\ 8(r_5)$ $\quad$ $//\text{mem}[r_5 + 8] = r_5$

The first instruction writes the contents of r_2 into the memory location at the address $r_4 + 4$ and the second instruction writes the contents of r_5 into the memory location at the address $r_5 + 8$. If these are local accesses, r_4 will be sp+offset1 and r_5 will be sp+offset2 where sp is the stack pointer and offset1 and offset2 are the offsets of r_4 and r_5 from the stack pointer. Therefore, the first memory store will be to the address sp+offset1+4, while the second will be to the address sp+offset2+8. If, for instance, offset1 and offset2 are found to be '12' and '8', respectively, then the two renamed values for the target operands will be m_0^1 and m_0^2, respectively: in other words, both memory stores will be to the same memory location. On the other hand, if, for instance, offset1 and offset2 are found to be '12' and '4', respectively, then the two renamed values will be m_0^1 and m_1^1, instead. In other words, the two memory stores will be to distinct memory locations. However, if the values of offset1 and offset2 cannot be determined statically, then the two renamed values will be m_0^1 and m_1^1 (i.e., we will name the written values as if they were written into distinct memory locations) and the physical locations indexed as m_0 and m_1 will be behaved as if they might coincide with any physical memory location. □

5.3.2 Determination of reaching definitions

Reaching definitions at a program partition point are determined by *Find_Reaching_Defs()* which is called by *Gen_RIG()* on line 4 of Listing 3. *Find_Reaching_Defs()* finds reaching definitions in a program partition using the labels on the forward edges of the PCFG of that program partition. Therefore, the RCG algorithm labels all the forward edges of the PCFG under consideration prior to reaching definition determination.

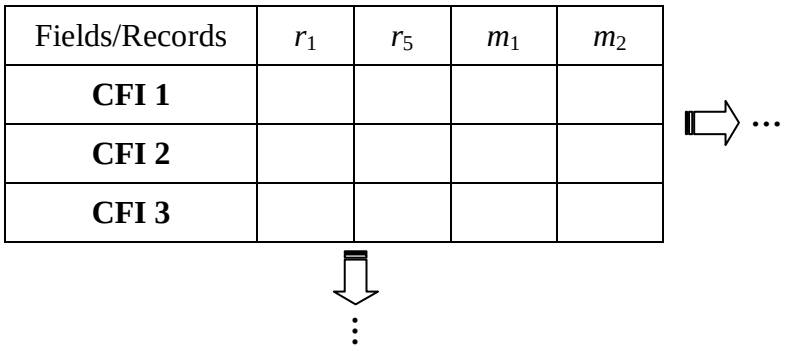


Figure 16: The renaming table structure.

To determine reaching definitions, *Find_Reaching_Defs()* should associate all the definitions encountered during the analysis of a program partition with the locations where those definitions are encountered. Since at most one definition can reach a point from an innermost control flow region, it is sufficient to associate a definition with the innermost control flow region in which that definition is made. For this purpose, a table called the *renaming table* is kept by *Find_Reaching_Defs()* (Figure 16). The renaming table has a record for every physical location (e.g., r_1, r_2, m_1, ...) that has been modified in a program partition up to

the instruction currently being analyzed. As more locations are modified, more records are added to the renaming table dynamically. Every record in the renaming table has a field for each CFI produced in a program partition 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:

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

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

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 program partition). According to the above rule, a renamed value generated within BB_i is written into the renaming table fields that correspond to the CFIs spanned by the labels on all incoming forward edges of BB_i.

However, applying rule (1) alone does not handle everything necessary for the determination of reaching definitions as explained. In addition to rule (1), *Gen_RIG()* performs two more actions: First, as stated in Section 5.3.1, 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 rule (1) deletes the entries in the same field(s) of the records belonging to other memory locations. Second, as mentioned before in Section 5.1, the edge-labeling algorithm allows the assignment of the same labels to distinct edges in a PCFG. This happens when distinct edges merge together at a confluence point in the PCFG, and after that, they diverge again. If there are two renamed values of a variable where one of the renamed values is given before a confluence point in the PCFG and the other is given after the confluence point, the latter may overwrite the former in the renaming table. This is because both of the renamed values might have to be entered into the same fields due to the assignment of same labels to the edges before and after the confluence point. Consequently, at a point where both definitions reach there statically, the latter definition might hide the former definition. In order to prevent this situation, when the analysis reaches a confluence point P in the PCFG of a program partition, *Gen_RIG()* combines the distinct definitions of a variable reaching P under a new *pseudo definition*. The pseudo definition is renamed as any other ordinary definition and is entered to the renaming table fields that correspond to the CFIs spanned by the labels on all the forward edges joining at P. However, as will be described in the next subsection, the combined reaching definitions are not completely thrown away but are represented by the pseudo definition in another data structure instead.

At a loop header block where a backward edge joins with a forward edge, *Gen_RIG()* delays the generation of the pseudo definitions due to the confluence of these edges until the whole loop is analyzed by *Gen_RIG()*. However, since backward edges are not labeled, edge labels cannot be used directly to find the loop carried definitions. Therefore, at the end of each pass over a loop body, *Gen_RIG()* carries the definitions reaching the end of the loop

tail block to the target of the backward edge of the loop. The pseudo definitions are similar in concept to the pseudo assignments of ϕ-functions in the SSA form generation; however, in the RCG algorithm, no prior search for the places of the ϕ-functions takes place [22].

Finally, 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, the 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 (we speak of a unique 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 and are represented by that pseudo definition). Let us now give an example of how reaching definitions are determined using edge labels and the renaming table.

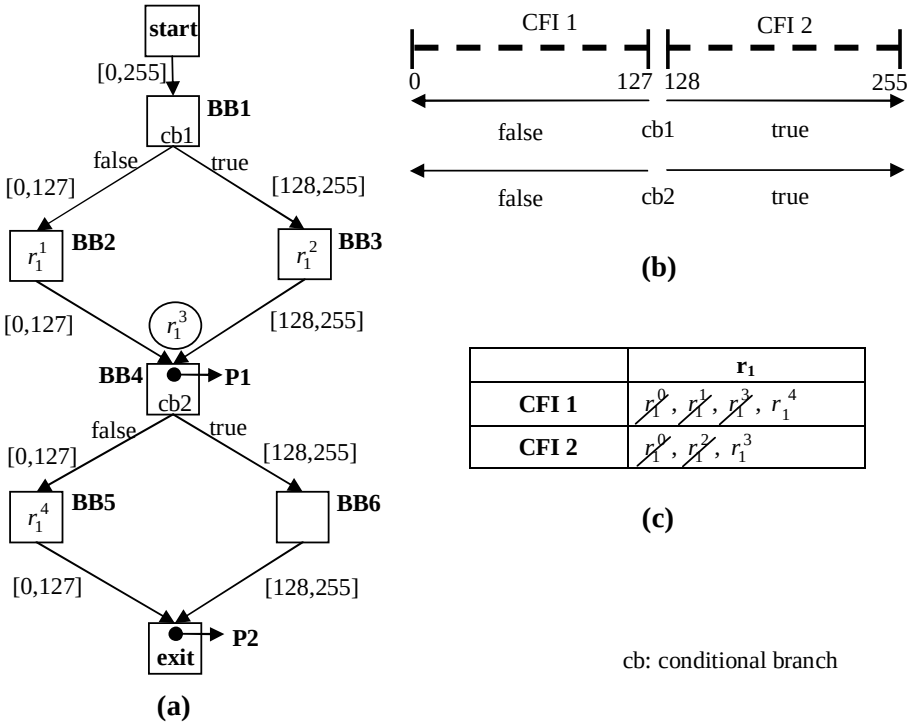


Figure 17: (a) A simple PCFG. (b) Corresponding CFIs. (c) Corresponding renaming table.

Example 5.6 *Determination of reaching definitions*: Consider the PCFG in Figure 17(a). Suppose that the RCG analysis is currently at the program point shown as **P2** in Figure 17(a) and we want to determine reaching definitions of register r_1 at **P2**. The CFIs and the renaming table generated for this PCFG are shown in Figures 17(b) and 17(c), respectively (note that the renaming table shows the entries that are generated up until the current point **P2**). For clarity, overwritten entries are also shown in the renaming table. When the analysis reaches the definition in BB2, a new value, r_1^1, is generated for r_1 and is entered into the renaming table field which corresponds to the CFI spanned

by the label on the incoming edge of BB2: CFI 1. Same operation is repeated for the definitions in BB3 and BB5 with the corresponding renamed values r_1^2 and r_1^4, respectively. When the confluence point **P1** is reached, *Gen_RIG()* combines definitions of r_1 reaching **P1** under a pseudo definition that is renamed as r_1^3, and then *Gen_RIG()* enters r_1^3 into the renaming table fields which correspond to the CFIs spanned by the labels on the joining edges at **P1**: CFI 1 and CFI 2. When **P2** is reached, *Gen_RIG()* queries the renaming table and finds the entries corresponding to the CFI fields spanned by the labels on the incoming edges at point **P2**. The entry corresponding to the left incoming edge (that of CFI 1) designates that the reaching definition of r_1 via the left incoming edge is r_1^4. On the other hand, the entry corresponding to the right incoming edge (that of CFI 2) designates that the reaching definition of r_1 via the right incoming edge is r_1^3 which represents r_1^1 and r_1^2 together. These definitions are indeed definitions of r_1 reaching point **P2**. However, note that if r_1^1 and r_1^2 were not combined under the pseudo definition r_1^3, r_1^4 would hide the reaching definition r_1^1 at **P2** since r_1^1 would have already been overwritten by r_1^4. □

5.3.3 Recovery of a destroyed variable

After finding the reaching definition for a variable that is modified by an instruction α, *Gen_RIG()* generates a RIG which reverses the effect of α by recovering the reaching definition found for the variable. This recovery is handled by the help of a *directed acyclic graph* (*DAG*), $DAG=(N,E)$. *Gen_RIG()* adds nodes and edges to the DAG both for the renamed values of the operands of α (or for the definitions made and used by α) and for the pseudo definitions which are generated at the confluence points. These nodes and edges together specify the relationship (or the data dependency) of a destroyed reaching definition with the other definitions generated in the program partition. Using this relationship, *Gen_RIG()* can recover the reaching definitions of the variable modified by α. The sets N and E of DAG 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.

Therefore, a node is inserted into the DAG for each definition in the program partition under consideration. Multiple definitions of a variable statically reaching a confluence point are merged under another node in the DAG: the node of the pseudo definition that represents those multiple statically reaching definitions. At a later confluence point in the program partition, a pseudo definition of a variable may again be merged with other pseudo or normal definitions of that variable reaching that confluence point.

Gen_RIG() also applies some annotations on particular nodes and edges in the DAG to provide 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 program partition point. Let us name the point just before and after α_{dest} as P and P', respectively. In order to recover D, *Gen_RIG()* tries to find the reaching definition of V at point P by calling *Find_Reaching_Defs()* at line 4 of Listing 3 (remember that in case there are multiple reaching definitions of V at point P, these definitions are represented by a unique pseudo definition 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 5.3.2). In this case, *Gen_RIG()* recovers D by generating state saving instructions. If a definition can be found, on the other hand, *Gen_RIG()* 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', *Gen_RIG()* can recover n_i by generating the appropriate 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 (i.e., by the redefine technique). On the other hand, if (b) holds, n_i can be recovered at P' by extracting n_i out of β (i.e., by the extract-from-use technique). 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 recovery of a node requires the knowledge

of the value of an external input of the program partition under consideration, *Gen_RIG()* generates state saving instructions to recover the killed node.

5.4 Putting it all together

In this section, we summarize the detailed operations of the RCG algorithm presented throughout Section 5.

To determine the intra-partitional control flow, the RCG algorithm uses predicate expressions which are determined for each edge coming to a confluence point (see Section 4.2). Predicate expressions for forward edges coming to a confluence point are determined by using the labels assigned to those edges (see Section 5.1). The predicate expressions for backward edges coming to a confluence point, on the other hand, are determined by the loop counter approach introduced in Section 5.2.

The RCG algorithm generates a RIG for an instruction α such that the RIG recovers the variable(s) directly modified by α. In order to recover a directly modified variable, the RCG algorithm uses a DAG (see Section 5.3.3). First, the RCG algorithm finds in the DAG the node for the reaching definition of the directly modified variable (for the determination of this node, see Section 5.3.2). Since α overwrites this definition, the node of this definition is killed. Then, in the DAG, the RCG algorithm constructs nodes and edges for the operands of α. Finally, the RCG algorithm tries to recover the killed node by using the other available nodes in the DAG (i.e., the nodes that have been constructed for the instructions scanned before α). For a loop, the RCG algorithm should not use the available nodes that are constructed for the instructions outside of the loop. If the only available nodes that can be used for the recovery of the killed node are the nodes that are constructed for the instructions outside of the loop, the RCG algorithm postpones the recovery to the next iteration of the loop provided that the total passes over the loop will not exceed three. If the loop has already been traversed three times, the RCG algorithm generates a RIG which employs state saving (see Section 4.7.1).

Let us illustrate the generation of an instruction-level reverse program with the example program shown in Figure 18.

Example 5.7 *Instruction-level reverse program generation*: Figures 19 and 20 show the renaming table and the DAG, respectively, that are constructed after two passes over the loop body (excluding the first pass over the whole program to generate the PCFG the CFIs and the CG) in the PCFG of Figure 18. The renaming table shows the analysis timestamps adjacent to a renamed value when that renamed value is generated (timestamps are shown in parentheses in Figure 18). The timestamp value increments by one after each instruction in a program partition is scanned. For clarity, the overwritten entries are again shown in the renaming table (Figure 19).

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 which correspond to the CFIs spanned by the incoming edge label [0,255]. Then, the newly generated value of r_4, r_4^1, is entered into the same fields according to the rule described in Section 5.3.2: the result can be seen in all the $r_4^1(2)$ entries in Figure 19. Next, a node 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

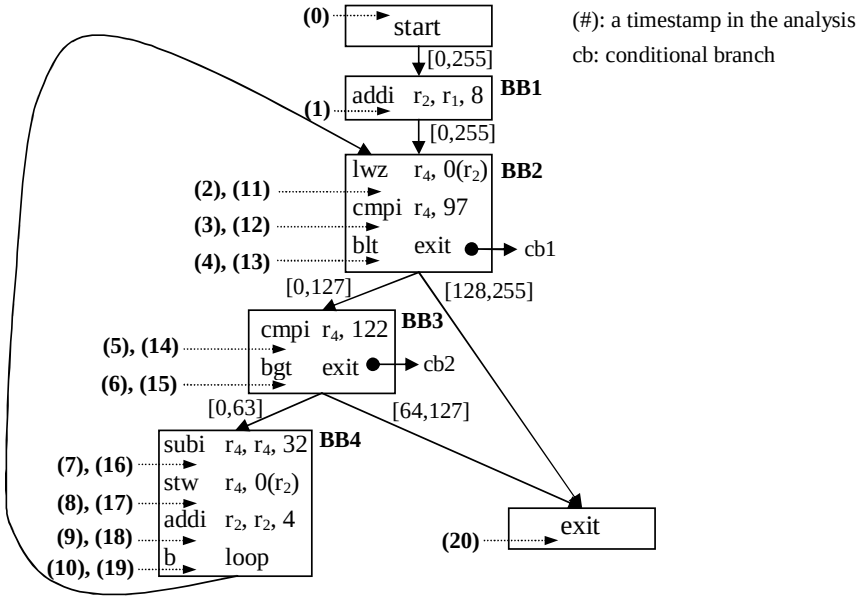


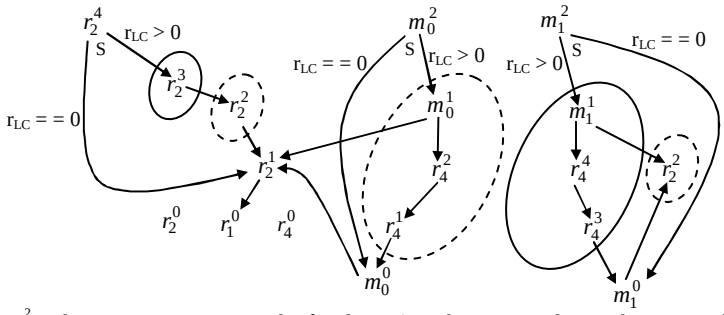
Figure 18: An example PCFG.

	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 19: The renaming table for the PCFG of Figure 18.

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 4.7.1 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 5.3.3 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. 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 "$addi\ r_2,\ r_2,\ 4$" (the instruction "$addi\ r_2,\ r_2,\ 4$" is found by the address annotation on r_2^2). Now, condition (b) given in Section 5.3.3 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 as explained in Section 4.7.1. Similar steps are followed for the generation of the RIGs for the other instructions as well.

After generating a RIG, the RCG algorithm connects the RIG to the previously generated RIGs by the function *Combine_RIGs()* (see line 14 of Listing 1). Figure 21 shows the modified code on the left



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 20: The DAG for the PCFG of Figure 18.

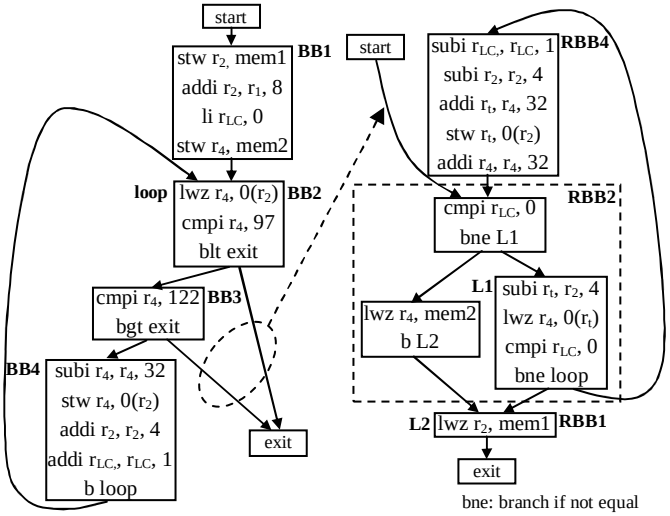


Figure 21: The original PCFG (left) and the reverse PCFG (right).

and the corresponding reverse code on the right. As explained before in Section 4.8, RIGs are placed in bottom-up order, and at the boundaries of the BBs, the edges of the original PCFG are simply inverted by generating the appropriate branch instructions in the reverse code. Consequently, a join point of edges in the original PCFG typically becomes a fork point of edges in the reverse PCFG, and vice versa. Note, however, that in this example, since the reverse of BB3 in Figure 21 happens to be empty (since BB3 does not include any instruction that directly modifies a register or a memory location), the inverted versions of the two incoming edges of the **exit** block in the original PCFG go to the same point in the reverse PCFG. 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 5.2 would be inserted at the end of the **start** block in the reverse code. □

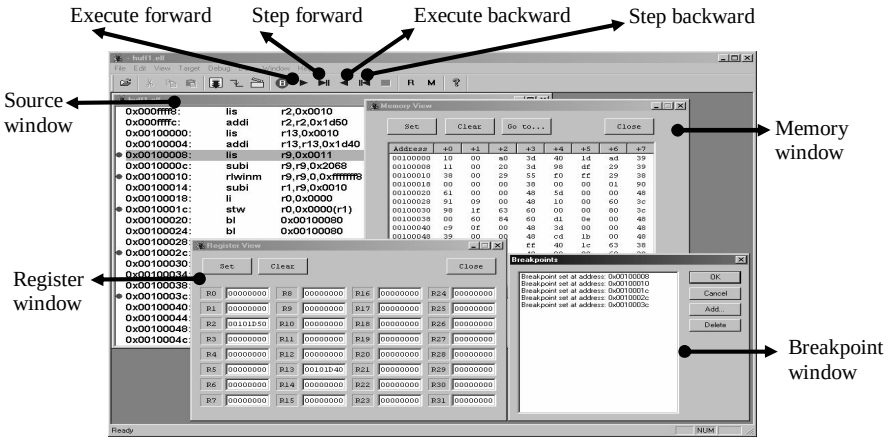


Figure 22: The GUI of the debugger tool.

6 Experimental Results

We tested the RCG algorithm on an evaluation board with a PowerPC (MPC860) processor. In order to test reverse execution on a debugging session, we implemented a low-level debugger tool with a graphical user interface (GUI) that provides debugging capabilities such as breakpoint insertion, single stepping, register and memory display (Figure 22). The debugger runs on a PC with Windows 2000. The PC is connected to the PowerPC board via a *Background Debug Mode* (*BDM*) interface [21].

The benchmark programs we used for our experimentation are a Fibonacci number generator (FNG) with 100 iterations, a selection sort (SSort) with 10 inputs, a 3 by 3 matrix multiplication (MMult) and a random number generator (RNG) with 100 iterations. FNG generates a Fibonacci series which includes 100 numbers. FNG does not write the generated numbers into memory, but calculates the numbers in a local variable. SSort sorts 10 numbers which are input to SSort in an array. SSort is an in-place sort algorithm. In other words, the numbers in the input array are sorted within the input array. This means that SSort does not allocate any additional temporary storage to sort the input data. MMult multiplies two 3 by 3 integer matrices that are input to MMult as arrays and writes the resulting matrix into another array. Finally, RNG generates 100 pseudo random numbers in a sequence. Similar to FNG, RNG does not use main memory to keep the generated numbers. All of the benchmarks are written in the C programming language. In order to compile the benchmarks for the PowerPC 860, we used a compiler from Tasking, Inc. [27]. Note that we compiled each benchmark using standard optimizations such as common subexpression elimination, constant propagation, constant folding, dead code elimination, strength reduction and global register allocation. Therefore, in our experimentation, the RCG algorithm generates an instruction-level reverse program for each benchmark by using optimized assembly code as input to the RCG algorithm. Table 1 depicts the size of each benchmark in terms of the total number of lines of C and the total number of assembly instructions.

In order to compare the performance of the RCG algorithm against the previous state saving techniques, we had to expand the previous techniques to support instruction-level reverse execution. Some of the previous techniques introduced in Section 3 are not applicable for

Table 1: The sizes of the benchmarks.

	FNG	SSort	MMult	RNG
#C lines	12	16	18	14
#assembly instructions	15	37	59	35

instruction-level reverse execution at all (e.g., source code transformation). The applicable ones, once expanded to support instruction-level reverse execution, are converted into either saving the modified processor state before each instruction (i.e., incremental state saving) or saving the modified processor state before each destructive instruction (i.e., incremental state saving for destructive instructions).

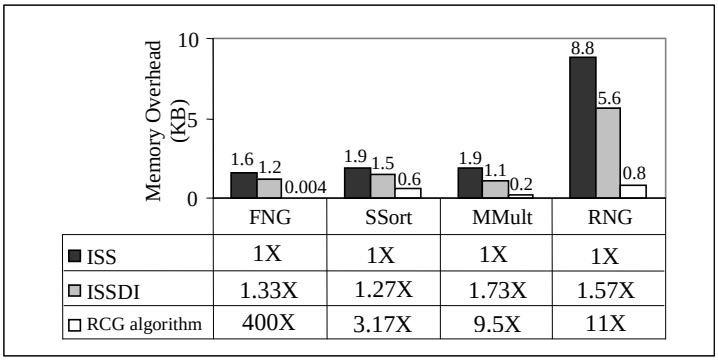
Table 2: Memory overheads.

	FNG	SSort	MMult	RNG
ISS	1.6 kB	1.9 kB	1.9 kB	8.8 kB
ISSDI	1.2 kB	1.5 kB	1.1 kB	5.6 kB
RCG algorithm	0.004 kB	0.6 kB	0.2 kB	0.8 kB

Table 3: Time overheads.

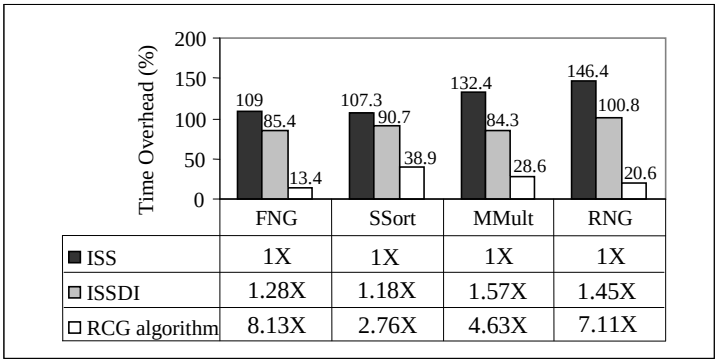
	FNG	SSort	MMult	RNG
ISS	109 %	107.3 %	132.4 %	146.4 %
ISSDI	85.4 %	90.7 %	84.3 %	100.8 %
RCG algorithm	13.4 %	38.9 %	28.6 %	20.6 %

Tables 2 and 3 show memory and time overhead results of the RCG algorithm, the ordinary incremental state saving (ISS) and incremental state saving for only destructive instructions (ISSDI). The memory overhead measurements were performed in the following way: For ISS and ISSDI, we calculated the program points where state saving is needed with each benchmark and we instrumented each benchmark with memory store instructions which save state at the calculated points. Then, we applied the RCG algorithm to each original benchmark in order to obtain the modified benchmarks instrumented with state saving instructions at necessary points for the RCG algorithm as well. For time overhead measurement, we used the decrementer counter of the PowerPC 860 processor (the PowerPC 860 provides a decrementer counter which is decremented by one at a certain number of processor cycles). First, we ran each benchmark without any instrumentation and noted the execution time (in number of processor cycles) using the change in the decrementer counter. Then, we ran modified benchmarks instrumented only with the necessary state saving instructions and noted the execution time (in number of processor cycles) in the same



	FNG	SSort	MMult	RNG
ISS	1X	1X	1X	1X
ISSDI	1.33X	1.27X	1.73X	1.57X
RCG algorithm	400X	3.17X	9.5X	11X

Figure 23: Memory overhead comparison.



	FNG	SSort	MMult	RNG
ISS	1X	1X	1X	1X
ISSDI	1.28X	1.18X	1.57X	1.45X
RCG algorithm	8.13X	2.76X	4.63X	7.11X

Figure 24: Time overhead comparison.

way. Finally, we calculated the time overhead for each benchmark by taking the differences between the noted execution times with instrumentation and the noted execution times without instrumentation.

Figures 23 and 24 show memory and time overhead comparisons, respectively, between the RCG algorithm, ISS and ISSDI. The results indicate that the RCG algorithm achieves from 3.17X to 400X and from 2.5X to 300X reduction in memory overhead as compared to ISS and ISSDI, respectively (Figure 23). Furthermore, the RCG algorithm achieves an average of 5.7X and 4.1X reduction in execution time of the benchmarks when compared to ISS and ISSDI, respectively (Figure 24). For the RCG algorithm, the relatively higher memory and time overheads that result from the measurements with SSort as compared to measurements with the other benchmarks are mainly due to ambiguous memory stores SSort uses. These ambiguous memory stores happen because the individual array elements that SSort overwrites during a sort operation depends on the initial ordering of the array elements. On the other hand, memory and time overheads encountered with the RCG algorithm during the execution of FNG only come from the loop counter inserted within the FNG loop. For this reason, a much bigger (300X - 400X) overhead reduction is achieved with the RCG algorithm as compared to previous approaches.

7 Conclusion

In this report, a new reverse execution methodology for programs is introduced. To realize reverse execution, the methodology generates a reverse program from an input program by a static analysis at the assembly level. The 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 generation of the reverse program is performed from the assembly instructions of a program, the methodology introduced in this report 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 full speed of the underlying hardware.

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*, pages 13–25, 1993.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, 8(3):21–26, May 1991.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, MA, 1986.

[4] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):194–206, December 1991.

[5] B. K. Bhargava. Concurrency control in database systems. *Knowledge and Data Engineering*, 11(1):3–16, 1999.

[6] M. R. Birch, C. M. Boroni, F. W. Goosey, S. D. Patton, D. K. Poole, C. M. Pratt, and R. J. Ross. Dynalab. *ACM SIGCSE Bulletin*, 27(1):29–33, March 1995.

[7] 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.

[8] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[9] 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.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] S. I. Feldman and C. B. Brown. IGOR: A system for program debugging via reversible execution. In *Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.

[12] J. Fleischmann and P.A. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, pages 50–58, 1995.

[13] R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, October 1967.

[14] R. M. Fujimoto. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation International*, 6(3):211–239, July 1989.

[15] F. Gomes. *Optimizing Incremental State Saving and Restoration*. PhD thesis, Department of Computer Science, University of Calgary, 1996.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[17] J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[18] D. A. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[19] Y.-H. Lee and K. G. Shin. Design and evaluation of a fault tolerant multiprocessor using hardware recovery blocks. *IEEE Transactions on Computers*, 33(2):113–124, February 1984.

[20] B. P. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 135–144, 1988.

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

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

[23] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 313–325, 1994.

[24] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[25] R. Sosic. History cache: Hardware support for reverse execution. *Computer Architecture News*, 22(5):11–18, December 1994.

[26] Steven Stolper. Questions and answers about the Mars Pathfinder, October 1997. http://www.quest.arc.nasa.gov/mars/ask/about-mars-path/.

[27] Tasking Inc. *Tasking C/C++ Compiler Datasheet*, 2001. http://www.tasking.com/products/PPC/ppc-ds21.pdf.

[28] 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.

[29] R. Wismuller. Debugging of globally optimized programs using data flow analysis. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 278–289, 1994.

[30] M. V. Zelkowitz. *Reversible Execution as a Diagnostic Tool*. PhD thesis, Department of Computer Science, Cornell University, 1971.

APPENDIX A

We have already stated that the generated reverse code only recovers memory or register values that are directly modified by the instructions and have explained how this is performed in the report. The remaining memory and register values that are not recovered by the generated reverse code are those values that are indirectly modified by the instructions. In this Appendix, we answer how we take care of the effects of indirectly modified memory and register values.

A value modified by an instruction is *directly* modified if the memory location or the register holding the value appears as an operand of the instruction; otherwise, the value modified by the instruction is *indirectly* modified. As an example, while the target operand of an "xor" instruction is directly modified by the "xor" instruction, a branch condition register may be indirectly modified by a "compare" instruction even if the branch condition register may not be an operand of the "compare" instruction.

Let us designate the set of instructions of a processor P with I. We define a set, say E ($E \subset I$), of instructions of P such that the outcome of an instruction in E does not depend on any indirectly modified memory location or register but only on that instruction's source operands which are directly modified by other instructions in I. The set of instructions outside of E, designated as E' ($E' \subset I$), on the other hand, are affected by indirectly modified memory and/or register values.

Example A.1 Consider ordinary integer addition and conditional branch instructions. The outcome of an ordinary integer addition instruction such as "*add* r_1, r_2, r_3" in a program is only affected by the values of r_1 and r_2 both of which are directly modified by other instructions in the program. Therefore, an "add" instruction is an element of E. On the other hand, the outcome of a conditional branch instruction such as "bne target" may depend on the value of a branch condition register which might be indirectly modified by a compare instruction. Therefore, a conditional branch instruction may be included in E'. □

Thus, we can omit the reverse code generation for the recovery of indirectly modified values if we can correctly undo the instructions in E'.

Therefore, the instructions in E' are specially treated as follows: Let us assume that the outcome of an instruction α depends on the value V of an indirectly modified memory location or register. Also, assume that an instruction, say β, computes V indirectly. Then, whenever α is to be reverse executed alone, the debugger tool reevaluates V by re-executing the instruction β in the background. If the instruction β has dependencies on other memory and/or register values, those register and memory values are recovered into temporaries prior to reevaluation of V. Let us explain this in the following example:

Example A.2 Consider the following instruction sequence (the numbers in the parentheses show the program points):

(1)
cmp r_{12}, *100*
(2)
bg *L1*
(3)

The outcome of the conditional branch instruction depends on the value of the branch condition register which does not appear as a source operand of the conditional branch instruction and which is indirectly modified by the compare instruction. Whenever the programmer reverse executes the program from point (3) to point (2) (i.e., the conditional branch instruction is reverse executed but the compare instruction is not), the debugger tool re-executes the compare instruction in the background. This guarantees that when the program is forward executed from point (2) on, the outcome of the conditional branch instruction will always be the same, even if the value of the branch condition register has been modified prior to reverse execution. □