# ASSEMBLY INSTRUCTION LEVEL REVERSE EXECUTION FOR DEBUGGING

A Thesis
Presented to
The Academic Faculty

by

## Tankut Akgul

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
April 2004

# ASSEMBLY INSTRUCTION LEVEL REVERSE EXECUTION FOR DEBUGGING

Approved by:

Professor Vincent J. Mooney III, Committee Chair

Professor Richard J. LeBlanc, Jr.

Professor Vijay K. Madisetti

Professor Santosh Pande

Professor Raghupathy Sivakumar

Date Approved: 7 April 2004

*To my mother Betul Akgul*

*and*

*my father Kamuran Akgul*

# ACKNOWLEDGMENTS

I am grateful to everyone who made this Ph.D. thesis possible. First and foremost, I owe special thanks to my advisor, Professor Vincent J. Mooney III, for his patience and guidance from the very beginning until the end. Second, I am especially grateful to Professor Santosh Pande for his valuable suggestions and guidance. Third, I would like to thank my committee members Professor Richard J. LeBlanc, Professor Vijay K. Madisetti and Professor Raghupathy Sivakumar for their constructive criticism which helped me improve this dissertation. I would also like to thank everyone in the Hardware/Software Codesign Group at Georgia Tech for their important feedback. Especially, the good times I had with my close friends Mohamed Shalan and Pramote Kuacharoen helped me bear all these years spent earning a PhD. Finally, I give special thanks to my wife Bilge whose love and support enabled me to complete this work.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

Many of the bugs in programs show their effects much later in program execution. For this reason, even the most careful programmers equipped with the state-of-the-art debuggers might well miss the first occurrence of a bug and thus might have to restart the program being debugged. Furthermore, for difficult to find bugs, this process might have to be repeated multiple times. However, every time a restart occurs, parts of a program that already executed without errors have to be re-executed unnecessarily. These unnecessary re-executions constitute a significant portion of the debugging time.

Reverse execution can be defined as a method which recovers the states that a program attains during its execution. Therefore, reverse execution eliminates the need for repetitive program restarts every time a bug location is missed. This potentially shortens debug time considerably.

Conventional techniques for recovering a prior state rely on saving the state into a record before the state is destroyed. However, state saving causes significant memory and time overheads during execution of programs.

This thesis presents a new approach which, for the first time ever (to the best of the author's knowledge), achieves reverse execution at the assembly instruction level on general purpose processors via execution of a "reverse program." A reverse program almost always regenerates destroyed states rather than restoring them from a record and provides assembly instruction by assembly instruction execution in the backward direction. This significantly reduces state saving and thus decreases the associated memory and time costs of reverse execution support.

Furthermore, this thesis presents a new dynamic slicing algorithm that is built on top of assembly instruction level reverse execution. Dynamic slicing is a technique which isolates the code parts that influence an erroneous variable at a program point. By the help of dynamic slicing, programmers can concentrate on the parts of programs that are actually related to bugs.

Similar to reverse execution, conventional dynamic slicing methods also require runtime information in the form of program execution trajectories. An execution trajectory captures the control flow information of a program. The algorithm presented in this thesis achieves dynamic slicing via execution of a "reduced reverse program." A reduced reverse program is obtained from a full reverse program by omitting the instructions that recover states irrelevant to the dynamic slice under consideration. This provides a reverse execution capability along a designated dynamic slice only. The use of a reduced reverse program for dynamic slicing removes the need for runtime execution trajectories.

The methodology of this thesis 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 up to 2206X reduction in runtime memory usage, up to 403X reduction in forward execution time *overhead* and up to 2.32X reduction in forward execution time for the tested benchmarks. Measurements on the selected benchmarks also indicate that the dynamic slicing method presented in this thesis can achieve up to six orders of magnitude (1,928,500X) speedups in reverse execution.

# CHAPTER I

# INTRODUCTION

## 1.1 Motivation and Aims

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

Despite today's computer-aided and automated technologies, debugging is still most effectively performed by a runtime interaction with the program under consideration. In this way, a programmer can see how a program actually behaves given a set of inputs and thus can evaluate the anomalies faster.
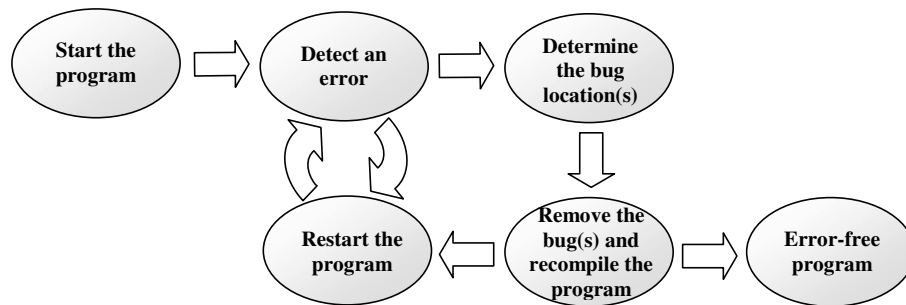
**Figure 1:** A typical debugging cycle.

A typical debugging cycle that many programmers go through is shown in Figure 1. 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 as shown in Figure 1. However, every time a restart occurs, parts of the program
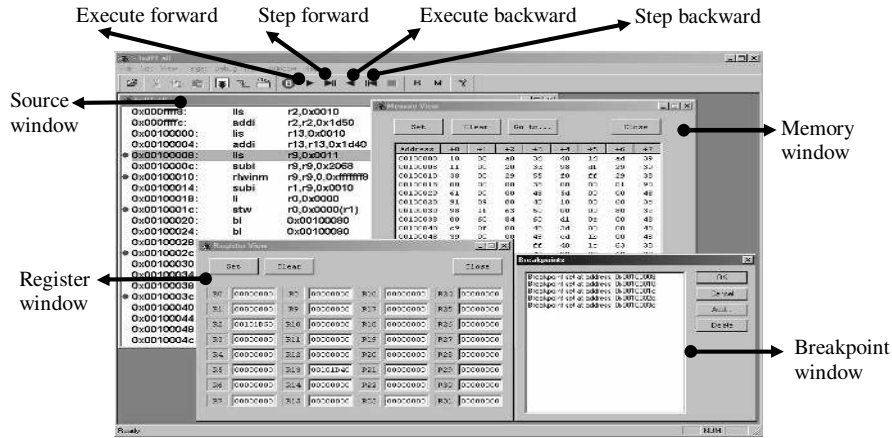
**Figure 2:** A sample view of windows from our reverse debugger tool.

that already executed without errors have to be re-executed unnecessarily. These unnecessary re-executions may constitute a significant portion of the debugging time. However, there is a very powerful technique which may speedup interactive debugging considerably. This technique is reverse execution.

Reverse execution provides the programmer with the ability to return to a particular previous state in program execution. When the programmer misses a bug location by executing a program too far, he or she can roll back the program to a point where the program state is considered to be correct and then re-execute from that point on without having to restart the program. This potentially reduces the overall debugging time significantly.

Figure 2 shows a sample view of windows from the debugger tool we have implemented. The programmer can load an assembly level program to the debugger tool and execute the program assembly instruction by assembly instruction both in the forward and the backward direction. At any desired point in execution (either in the forward or in the backward direction), the programmer can observe the values stored in registers and memory locations. Moreover, the programmer can set breakpoints at various places in the program and forward or reverse execute the program until those breakpoints.

2

A reader may ask, "Why is reverse execution at the assembly level important?" There are various reasons making reverse execution (or even forward execution) at the assembly instruction level available in a debugger. First, performance and memory constraints or lack of compiler support usually force 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 [44]. Furthermore, even for general purpose software, the source code may not be available at all. Therefore, during debugging of such software components, programmers have to be involved in assembly instruction level program execution.

Second, in implementing a language construct such as a pointer to an integer, sometimes the compiler generates assembly different from what the programmer expected. Similarly, compiler optimizations can move or remove instructions resulting in assembly code that does not match with the source code anymore, which makes it very difficult to debug at the source code level.

Finally, reverse execution at the assembly instruction level provides extremely fast backup capability in case the programmer executes one extra instruction too far, losing the whole program context leading to a bug.

Much like reverse execution, another powerful technique that may be used to speed up debugging is dynamic slicing. Dynamic slicing isolates the statements that influence the value of a variable at a program point in a specific execution instance of the program. In this way, the programmer can concentrate only on those statements that are relevant to the bug(s) in the program.

Therefore, it is desirable to implement both assembly instruction level reverse execution and dynamic slicing under a common framework that will help significantly reduce the time spent on debugging.

## 1.2  Problem Statement

*Note that in the rest of this thesis, the word "instruction" refers to an assembly instruction.*

An execution of an assembly 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 1.2.1** *Instruction level reverse execution:* Reverse execution of an assembly program $T$ running on a processor $P$ can be defined as taking $P$ from its current state $S_i$ to a previous state $S_j$ $(0 \leq j < i)$. The closest achievable distance between $S_i$ and $S_j$ 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 the state 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.

Similarly, dynamic slicing usually requires runtime control flow information from the execution of a program in terms of execution trajectories (see Section 2.2.2). As

programs can follow more than one control flow path, the actual path the program follows determines which instructions are redundant instructions that are irrelevant to a slice. This runtime information is a significant factor making traditional dynamic slicing methods costly in terms of memory usage.

Briefly, the problem addressed by this thesis is the achievement of time and memory efficient reverse execution and dynamic slicing. In particular, an approach resulting in a dramatic reduction in state saving for both reverse execution and dynamic slicing is sought.

Having discussed the problems associated with reverse execution and dynamic slicing, we next outline the approach and the contributions of this thesis.

## 1.3 Thesis Contributions

In this thesis, for the first time ever (to the best of the author's knowledge), an instruction level reverse execution methodology in software for general purpose processors is proposed via use of "reverse program generation." The proposed methodology is unique in the sense that it provides reverse execution at the assembly instruction level and yet still has reasonable memory and time overheads when the program is being executed.

In the proposed technique of this thesis, destroyed states are almost always regenerated instead of being restored from a previously saved record. This provides faster execution speeds and less memory space overheads as compared to traditional techniques.

This thesis also implements a new dynamic slicing algorithm on top of assembly level reverse execution. When a programmer realizes that a variable has an incorrect value at a certain program point, he or she can reverse execute the program along the corresponding dynamic slice only. Thus, instructions that are irrelevant to the bug(s) in the program are skipped and a faster return to the suspicious location can

be achieved. Most designers do not know *which* dynamic slice is needed *until* the bug appears. In this case, our approach is faster than re-running the application with the dynamic slice specified.

The dynamic slicing technique presented in this thesis not only speeds up reverse execution but also contains an advantage which is usually not offered by traditional dynamic slicing methods (see Section 2.2.2). Our technique can dynamically reconstruct the control flow information during reverse execution. Therefore, as opposed to traditional dynamic slicing methods, our technique does not require an execution trajectory to extract the control flow information from a program.

## 1.4   Thesis Organization and Roadmap

The thesis is organized as follows. Chapter 2 presents the related work. Chapter 3 gives an outline of our approach. Then, Chapter 4 explains the implementation of the proposed reverse execution technique and Chapter 5 fills in special implementation details. After explaining the reverse execution technique, Chapter 6 gives a summary of the presented work. Then, Chapter 7 discusses the extensions made to the presented algorithm with the addition of dynamic slicing support. Chapter 8 gives various experimental results. Finally, Chapter 9 concludes the thesis.

# CHAPTER II

# PREVIOUS WORK

The problem of how to acquire previously destroyed program states has been researched in several contexts. Similarly, there has been a considerable amount of work with regard to program slicing. In this chapter, we first present different techniques applied for reverse execution and then present the previous research efforts for program slicing.

## 2.1  Previous Work in Reverse Execution

Reverse execution research can be divided into two different categories. The first category is the application of pure state saving approaches to restore earlier states in program execution. The second category, on the other hand, is reverse execution by a combination of state saving and state regeneration techniques.

State regeneration introduced in the second category reproduces previously destroyed values and thus achieves state recovery without state saving. This helps reduce the amount of memory usage. However, the first category covers almost all of the research performed so far to achieve reverse execution of programs. Moreover, we see very limited forms of state regeneration techniques in prior research (see Section 2.1.2). On the contrary, state regeneration constitutes the basis of this thesis.

The following two subsections provide a summary of previous work in each category mentioned above.

### 2.1.1  Reverse Execution Techniques by Restore Earlier State

Zelkowitz provides a state restoration capability by inserting trace statements into the programming language [50]. Each trace statement includes an option that 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 for programs written in a high-level programming language [3]. Agrawal et al. statically associate each assignment statement with 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, 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.

Booth and Jones associate each variable in a program with a story tag which includes the history information about how the variable is computed [14]. When a variable is used in a computation, its story is added to the story of the computed variable. At the end of a particular execution, the programmer can trace back how a variable is computed by observing the story of the variable only. However, this approach may again cause very large memory usage as the variable stories are built by pure state saving.

State restoration is also applied in so-called replay techniques for efficient debugging of programs by either hardware [7, 43] or software [19, 35, 36, 40, 42]. In the replay technique, the state of a program is first saved infrequently during execution of the program, and then the program state that is not saved is reconstructed by

replaying the program in the forward direction. In hardware approaches, state saving is handled by hardware with little or no performance overhead but with inflexibility and high cost. 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 trace window where state is saved and not at an arbitrary program point. Trace windows may be fixed [19, 40] or variable [40, 36] in length and their sizes may either be determined by the programmer [40] or by a program analysis [40, 36]. For instance, in Miller and Choi's approach, trace windows which are called emulation blocks are usually chosen to be program regions that have well-defined entry points [36]. One such region is a subroutine. Generally, larger trace windows result in smaller traces but longer replay times.

Two other application areas of state restoration are optimistic or speculative computation [22, 24, 30] and fault tolerance [16, 34]. A computation is optimistic if incorrect computation is allowed during execution. Tasks executing in parallel usually have to block due to synchronization requirements on shared data. In optimistic parallel execution, tasks do not block on shared data and thus 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, state restoration for fault tolerance is performed by rolling back in case software errors occur, which is usually seen in places such as database transaction systems [10, 25].

Rolling back computations or transactions is usually achieved by periodic or incremental state saving. In periodic state saving [20], 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. If the checkpointing interval is reduced, 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 (ISS) [48], instead of recording the whole processor state, only the modified parts of a state are recorded at each checkpoint. However, in programs where the modified state space is large, memory and time overheads of incremental state saving might again exceed affordable limits.

State restoration 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 [12, 18]. Program animation constructs a virtual machine with a reversible set of instructions. Since these instructions are reversible, the program can be run backwards. However, since reversible instructions are usually constructed as stack operations, a significant amount of stack memory may be required in program animation.

Cook presents a state restoration method for Java programs [17]. A Java virtual machine is another example of a stack-based virtual machine. Most Java bytecode instructions pop the operands from the operand stack, operate on them and then push the resulting operands back into the operand stack. In his method, Cook keeps two main circular buffers for keeping the program state. The first buffer, the program counter circular buffer, keeps the program counter values. The second buffer, called the logging circular buffer, keeps the values in the operand stack that are destroyed as a result of a bytecode instruction. Then, Cook associates a reverse operation with each bytecode instruction. Basically, during a reverse operation, the program counter

is restored from the program counter circular buffer while the values in the operand stack are restored from the logging circular buffer. By using circular buffers, Cook bounds the memory requirement which otherwise would be huge due to logging of each modified operand. However, this also bounds the total number of bytecode instructions that can be reverse executed at a time and requires a runtime validation of whether there is enough data within a circular buffer for accomplishing a reverse execution request.

When compared to pure state saving saving approaches explained in this section, our reverse execution technique via generation of a reverse reverse program potentially reduces the runtime memory usage considerably. This is because most of the program state is regenerated by the reverse program. Moreover, regeneration of program states via a reverse program reduces the number of state saving instructions in the original code and thus provides faster forward execution times.

## 2.1.2    Previous Work in State Regeneration at the Source Code Level

Floyd makes use of state regeneration in the area of nondeterministic algorithms [21]. A nondeterministic algorithm is an algorithm that may come up with a different solution to a problem at each run of the algorithm. However, the solution is not reached by a random process but by incrementally constructing a path that 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.

Floyd achieves reverse execution by defining a reverse operation for each operation in a nondeterministic algorithm. However, a reverse operation without state saving can only be generated for reversible constructive operations. A constructive operation is an operation where the variable being modified (the target operand) is the same as one of the source operands. The operation "x = x + 1" is an example of a

constructive operation. On the other hand, a constructive operation is reversible only if there exists an operation that can fully recover the prior value of the target operand of the constructive operation. For instance, the prior value of the target operand of "x = x $\oplus$ 1" can be fully recovered by executing the same operation once more. Therefore, "x = x $\oplus$ 1" is reversible. On the other hand, although "x = x / 2" is constructive, this operation might not be reversible for the case where the target operand $x$ cannot always be fully recovered due to a possible precision loss in the computed result.

State regeneration finds its application in a limited sense in the area of debugging optimized code as well [1, 28, 49]. Hennessy introduces the term "currency" of a variable [28]. 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 according to 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 at a program point is current or not 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 [49].

Carothers et al. introduce another approach for optimistic parallel simulations [15]. This approach is source transformation. In source transformation, the source code (e.g., in C) is transformed to a reversible source code version. Then, from the reversible version a reverse source code is obtained. The reverse source code reverses the effect of the original source code. However, similar to Floyd's approach [21], Carothers et al. also apply state saving for all operations except reversible constructive ones. Consequently, the execution time and memory requirements of the reverse code are increased.

Biswas and Mall also generate a reverse program from a program given in the C programming language [13]. For constructive operations in C which use operands such as "*=" or "+=", they generate reverse statements that use the inverse operators. Thus, for instance, "*=" becomes "/=". However, similar to other approaches presented in this section, these constructive operations are the only cases where reverse execution is performed without state saving. Moreover, in cases where the underlying processor may truncate the result (such as an overflow condition), the correctness of the reverse execution is indeterminate. For all other operations, Biswas and Mall keep a trace file that holds the necessary state to reverse execute a C program. In their paper [13], Biswas and Mall state (referring to non-constructive assignment statements in the form of $a = b$ op $c$), "Thus, it is *not possible* to define an inverse for this [sic] type [sic] of assignment statements and the old value of the variable must be maintained in a trace file." In a similar quote, the DDD (a front-end for GNU GDB debugger) v3.2 manual on page 98 states, "DDD cannot undo what your program did. After a little bit of thought, you will find that this would be *impossible* in general." In this thesis, we show that even destructive assignment statements are indeed reversible without state saving.

The state regeneration techniques proposed in this thesis are applicable in wider range of situations than the state regeneration techniques presented in this section.

Therefore, state regeneration efficiency is potentially much higher in our techniques when compared to previous techniques. This results in larger savings in terms of memory and time costs.

### 2.1.3 Previous Work in State Regeneration at the Assembly Instruction Level

We have made an exhaustive literature survey searching for a software approach that achieves reverse execution at the assembly instruction level. However, we could not find any such work. Therefore, this thesis might be the first work to achieve reverse execution at the assembly instruction level on general purpose processors by use of reverse program generation.

## 2.2 Previous Work in Program Slicing

Program slicing algorithms can be divided into two major types according to whether the slice extraction is performed at compile time (*static slicing*) or at runtime (*dynamic slicing*). In this section, we present a brief history of static slicing as well as dynamic slicing algorithms. An interested reader can find detailed surveys of slicing algorithms in [11, 27, 46].

### 2.2.1 Static Slicing

Program slicing was first proposed by Weiser [47]. His approach statically determines the program statements that potentially affect a variable at a certain statement. Here, the variable under consideration and the statement at which it is computed form the slicing criterion. Weiser computes slices from a control flow graph of the subject program. For this purpose, Weiser iteratively finds data flow information between control flow graph nodes. This information reveals at each control flow graph node the variables, called *relevant* variables, that transitively affect the computation of

the variable in the slicing criterion. Then, the slice is found as a combination of all program statements that compute the relevant variables in a program.

In Weiser's approach the extracted slices are executable. That is, the extracted slice is itself an executable subset of the original program. Also, the slices introduced by Weiser are backward meaning that the statements affecting the variable in the slicing criterion are found by a backward traversal of the control flow graph nodes starting from the node computing the variable in the slicing criterion.

Ottenstein and Ottenstein use a different intermediate representation to compute slices [41]. Instead of a control flow graph, they use a *program dependence graph* (*PDG*). The statements and the control flow predicates in a program constitute the vertices of a PDG, while the data and control dependence relationship between the vertices are represented by the edges of a PDG. Slicing is defined as a reachability problem in a PDG. The nodes that are reachable from the vertex that computes the variable in the slicing criterion are included in the slice. Later, Horwitz et al. introduce various modifications to the PDG approach to determine static slices of a program including multiple procedures or unconditional jumps such as goto statements [29].

Bergeretti and Carre introduce forward slicing [8]. Instead of looking for variables that might affect the variable in question (as in the case of backward slicing), forward slicing specifically looks for variables that *might be affected* by the variable in question. Therefore, as opposed to backward slicing, forward slicing traces the dependencies between variables in the forward direction.

Harman and Danicic propose a different approach to slicing [26]. They define a slice called an *effect minimal slice*. An effect minimal slice is a slice that preserves the effect of the original program on the variable in the slicing criterion, but that is not necessarily constructed as a subset of the original program. Harman and Danicic convert an imperative program to its functional equivalent and then perform algebraic transformations on the resulting functional program by using the properties

15

of functional languages. The transformations replace the statements in the original program with less number of equivalent statements that have the same overall effect on the variable in the slicing criterion. Therefore, a thinner static slice can be obtained for the variable in the slicing criterion.

The main drawback of static slicing is that it has to be conservative in making decisions that require runtime information of a program. For instance, if two statements statically affect the variable in the slicing criterion, but one of the statements actually is never executed in a particular debugging session, static slicing has to conservatively include both statements in the slice because the decision of whether the statements will be executed or not may not be made statically. Thus, static slicing usually provides slices larger than one could potentially obtain by using a dynamic slicing technique including the dynamic slicing technique proposed in this thesis.

### 2.2.2   Dynamic Slicing

As explained in the previous subsection, static slicing does not employ any runtime information and thus static slicing may compute large slices. This disadvantage of static slicing motivated Korel and Laski [31] to introduce dynamic slicing. Their approach incorporates runtime information to find the statements that *actually* affect a variable at a program point. Thus, the resulting slices are more compact and precise than the program slices proposed by Weiser [47]. First, the subject program is run and an execution history is obtained. The execution history tells which statements in the program actually execute. Then, using this information, statements upon which the variable in the slicing criterion is dynamically data or control dependent are extracted.

Agrawal and Horgan incorporate a dependence graph, which they call a *dynamic dependence graph* (*DDG*), to calculate dynamic slices of a program [4]. In the sense that a dependence graph is used, Agrawal and Horgan's approach can be considered

16

as an extension to the PDG based static slicing algorithms. Later Agrawal, DeMillo and Spafford present a solution based on DDGs for inter-procedural dynamic slicing of programs [2].

There has been research in forward computation of dynamic slices as well [9, 32]. In a forward dynamic slicing technique, a dynamic slice is obtained during execution of the program. Therefore, the runtime trace information is kept bounded. However, since forward slicing does not start from the instruction in the slicing criterion, it calculates all possible dynamic slices for all variables in a program. Therefore, forward dynamic slicing algorithms are usually slow.

In dynamic slicing techniques, the processing of a complete program trace also constitutes a large memory usage. Zhang et al. present an algorithm which keeps a record of the complete execution history and then processes only the necessary information in that record for the purpose of generating a particular dynamic slice [51]. This potentially reduces the memory cost of slice extraction.

The main drawback of traditional dynamic slicing methods is that these methods either use too much runtime information or are not performed in a demand-driven fashion starting from a specific dynamic slicing request. Moreover, the traditional dynamic slicing methods do not provide a way for reverse execution along a dynamic slice. However, the dynamic slicing method proposed in this thesis is demand-driven, can regenerate most of the control flow information required to extract the dynamic slice during reverse execution and provides a way to reverse execute the dynamic slices.

## 2.3   Summary

In this chapter, we presented various prior research in reverse execution and dynamic slicing. The reverse execution technique we present in this thesis usually causes much

less memory and time overheads than the prior reverse execution techniques. Moreover, as opposed to previous approaches, the dynamic slicing technique we propose reduces runtime memory usage and provides reverse execution along dynamic slices. We will eventually show the advantages of our technique over the previous techniques with quantitative comparisons in Chapter 8. In the next chapter, we present an overview of our approach for reverse execution at the instruction level. The implementation details of our reverse execution technique and the extensions for dynamic slicing support are explained in chapters subsequent to the next chapter.

# CHAPTER III

# OVERVIEW OF REVERSE CODE GENERATION

Our approach is mainly based on regenerating a previously destroyed state rather than restoring that state from a record. When state regeneration is not possible, however, we recover a destroyed state by state saving. Hereafter, we refer to the recovery of a state as *state restoration* if that recovery involves state saving. If there is no state saving involved in a particular state recovery, we refer to such recovery as *state regeneration.*

We achieve both state regeneration and state restoration by the help of a reverse program. Given an assembly program $T$ either written directly in assembly or compiled into assembly from a high level programming language, we generate a reverse program $RT$ from $T$ by a *static* analysis at the assembly instruction level. We call our algorithm which we use to generate a reverse program the *Reverse Code Generation* ($RCG$) algorithm.

The RCG algorithm can be divided into three main steps. These steps are program partitioning, reversing an assembly instruction and combining the reverse instructions. Figure 3 shows the flow of control between these three steps. The RCG algorithm first passes over the input program and partitions the input program (Step 1). Then, the RCG algorithm goes into a main loop where each program partition is read instruction by instruction in program order (lexical order). After an instruction $\alpha$ is read, the RCG algorithm generates the reverse of $\alpha$ (Step 2 in Figure 3) and combines the reverse with the rest of the reverse program generated so far (Step 3).

Step 2 and Step 3 are repeated until the end of a program partition is reached. When the end of a program partition is reached, the RCG algorithm moves to the next program partition and iterates over Step 2 and Step 3 for the instructions in the next program partition. These steps are repeated until all the partitions inside a program are processed and the end of the program is reached.
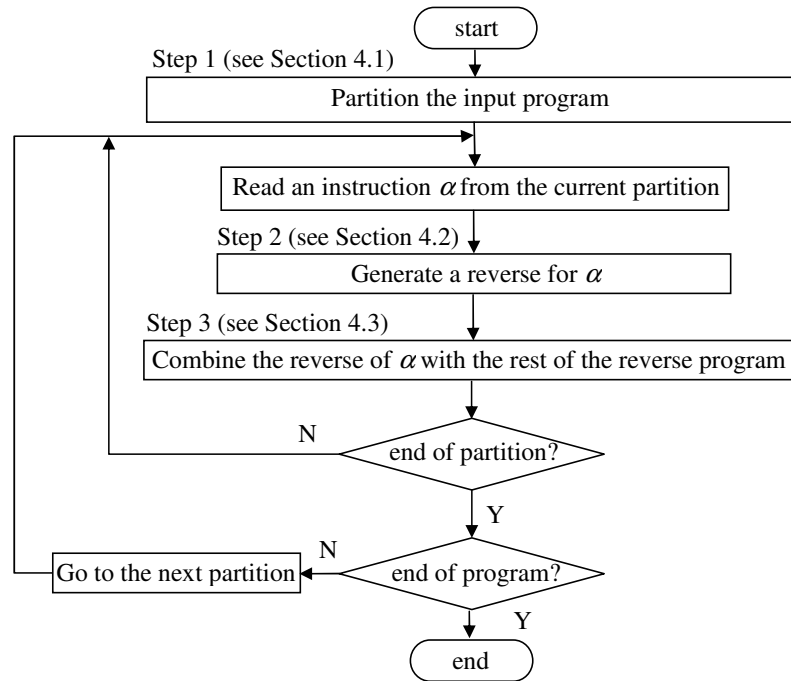


**Figure 3:** A high-level view of the RCG algorithm.

In the following sections, we first state our main assumptions for the applicability of the RCG algorithm and then discuss each RCG step briefly.

## 3.1 Preliminary Assumptions

We assume a certain set of inputs to be provided to the RCG algorithm. Let us first describe our assumptions regarding the inputs of the RCG algorithm and then explain under which conditions the RCG algorithm can apply the state regeneration techniques that will be presented in the rest of this thesis.

### 3.1.1 Inputs of the RCG Algorithm

Figure 4 shows the input of the RCG algorithm. First, the RCG algorithm takes as input either through a Graphical User Interface (GUI) or as a command line argument an assembly program $T$ without associated source code. We assume no symbol table nor any other compiler related information whatsoever. The assembly code can be hand-written or generated by a compiler. Moreover, compiler optimizations do not limit the applicability of the RCG algorithm. As will be shown in Section 8, the RCG algorithm can provide significant savings in execution time and memory usage even with highly-optimized programs as input. Also, we assume that the input program is single-threaded, is not self-modifying and does not change base addresses of program sections (such as the global data section or the stack) dynamically.

| Input program | | |
|---|---|---|
| li | $r_{11}$,0x1 | |
| cmpw | $r_8$,$r_{11}$ | |
| blt | 0x8024 | |
| add | $r_9$,$r_{12}$,$r_{10}$ | |
| ori | $r_{10}$,$r_{12}$,0x0 | |
| ori | $r_{12}$,$r_9$,0x0 | |
| addi | $r_{11}$,$r_{11}$,0x1 | |
| b | 0x8008 | |
| blr | | |
| ... | | |

Command line or GUI

RCG

Overflow/underflow switch ("-no-extract-from-use")
Number of maximum recursions ("-max-recursions=N")

RCG options
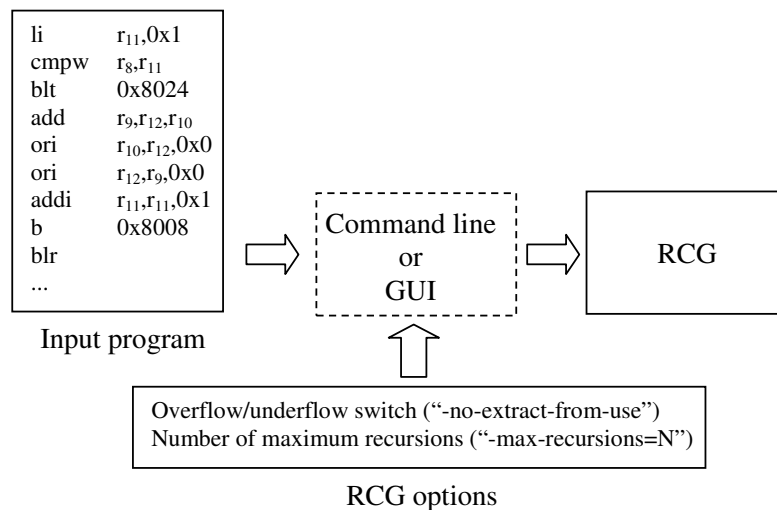
**Figure 4:** Inputs of the RCG algorithm.

In addition to the input program, the RCG algorithm accepts two options from the user (Figure 4). These options affect the reverse code generation process. First, in case of an overflow or an underflow during an operation to be reversed, a specific state regeneration technique, which we will name the extract-from-use technique and describe in Section 4.2.2, may not work properly. Therefore, in a program that may

trigger an overflow or an underflow, the user notifies the RCG algorithm by use of a command line switch ("-no-extract-from-use") or through a GUI that overflow or underflow may occur (see Figure 4); otherwise, if "-no-extract-from-use" is not specified in the command line input (or through a GUI), RCG assumes no overflow or underflow ever occurs under any situation (e.g., for any possible data input). The use of "-no-extract-from-use" option ensures that the reverse code functions correctly in cases where overflow or underflow may occur.

Second, the user specifies by use of a command line option "-max-recursions=$N$" (or through a GUI) the upper limit for how many times the RCG algorithm may recursively apply the state regeneration techniques. In this option, the maximum number of recursions allowed is specified by $N$. As will be explained in Section 4.2, the RCG algorithm usually applies the state regeneration techniques recursively because these techniques may not give a solution after only one iteration. However, each recursive application of state regeneration techniques increases the running time of the RCG algorithm. Therefore, the upper limit $N$ set by the "-max-recursions=$N$" option helps limit the time cost of the RCG algorithm. If an upper limit $N$ is not set, the RCG algorithm applies state regeneration techniques repetitively until a point where all paths to possible solutions are traversed. If no solution can be found along those paths, state saving is dictated.

Next, we state the assumptions we make regarding the reverse code generation process.

### 3.1.2 Assumptions for RCG

Now, suppose that an assembly program $T$ attains a series of states $S = (S_0, S_1, S_2, ...)$ during its execution on a processor $P$ where the distance between two consecutive states is one instruction. Now, assume that we can generate another program $RT$, the instruction level reverse program of $T$, such that when a specific portion of $RT$

is executed in place of $T$ when the current state of $T$ is $S_i$, the state of $T$ can be brought to a previous state $S_j$ $(0 \leq j < i)$. In other words, $RT$ recovers a previously destroyed state. 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 the execution of $T$ can be reversed by executing $RT$ in place of $T$.

To build a complete reverse program, the RCG algorithm uses pure static information. However, in order to reverse execute a program, dynamic information may be required as well. This dynamic information mainly consists of the values that cannot be recovered by state regeneration and thus are saved dynamically. The statically generated instructions inside the reverse program use this dynamic information to undo the original instructions. Therefore, in case of state restoration, $RT$ simply issues memory load instructions that restore the values already saved by state saving instructions inserted into $T$. In the case of state regeneration, $RT$ recovers a state without any state saving code in $T$.

However, in practice, it might be hard to implement such a program $RT$ which recovers 100% of the program state either by state regeneration or by state restoration. 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. For a more detailed discussion of indirectly modified state in an Instruction Set Architecture

(ISA) with a specific focus on PowerPC as a target example ISA, please see Appendix A.

2. Generally, writing a value to the program counter either by a branch instruction or by direct modification causes an immediate jump to the address location designated by the value written to the program counter. 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 (including exact cache state).

Therefore, regarding item (1) above, we define a program state $S' = (M',R')$ that 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 $T$). Assuming that we can generate an instruction level reverse program $RT$ for a program $T$ (an assumption which is true since we can always resort to state saving if necessary), we can recover all memory and register values that are directly modified by $T$. Note that some indirectly modified memory/register values that have an effect on $T$'s state may cause incorrect computation in rare cases if those values are not recovered properly. We propose two techniques that can be used to ensure correct computation in such cases. Therefore, by the help

of the two techniques we propose, we can even reverse any program that includes any combination of instructions that indirectly modify program state. For a detailed discussion of these two techniques, please refer to Appendix A.

Regarding item (2) above, since the program counter value carries important debugging information, we must 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. We associate the address of each instruction in $T$ with the beginning address of the corresponding portion in $RT$ that 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 $RT$.

Finally, regarding item (3) above, we treat memory as a unified abstract entity which keeps the values of high-level program variables. In other words, as long as the destroyed values of a variable can be retrieved, we do not distinguish between whether the variable is actually kept in processor cache or main memory. Consequently, undoing a memory store operation on a program variable only comprises recovering the previous value of the program variable but not the exact previous state of the processor cache or main memory. For instance, a variable may originally reside in main memory but not in the L1 cache; however, after the variable is destroyed and subsequently recovered, the variable might be brought into the L1 cache. Therefore, this process recovers the value of the variable, but does not restore the original state of the L1 cache.

In addition to the three items above, there are also two other assumptions we make. First, we assume that in case of an exception, the exception handler saves the program context such as the program counter and the register state just before the exception. Therefore, exceptions can be reversed by recovering the saved program

25

context. Second, we assume that the external inputs/outputs to/from a program (such as file I/O) are recovered by state saving.

Having discussed some of our initial assumptions and goals, we next illustrate the highlights of the three main RCG steps.

## 3.2  Program Partitioning

Many programming language related algorithms require a certain analysis which is used to extract information from the subject program. The same is true for the RCG algorithm. The control flow analysis and the reaching definitions analysis constitute the main analyses of the RCG algorithm.

When analyzing a program, it is essential to first select the program level at which to perform the analysis. Is the analysis to be carried out on the whole program at once or on individual functions separately or on even smaller regions such as basic blocks one at a time?

It is usually difficult to perform control flow analysis across indirect calls because an indirect call may be made to a statically unknown address. Therefore, in order to simplify the RCG algorithm, we prefer to restrict the control flow analysis to inside certain regions of code in which control flow can be statically determined. We name these regions *program partitions* (*PPs*). Across PPs, on the other hand, control flow information is dynamically recorded via a state saving technique that will be explained in Section 4.3.3.2.

For most instruction sets (e.g., PowerPC, x86 and ARM), PPs are delimited by indirect branches or "function call" instructions that may exist in the code. An indirect branch delimits a PP because an indirect branch instruction may direct the control to a statically unknown address. A function call instruction, on the other hand, delimits a PP because the control may return from the called function whose address may be statically unknown to the address following the function call

```
void main() {
    int z;
    z = foo(5);
    z = z + 1;
}

int foo(x) {
    int a, b, c = 3;
    b = x | 15;
    a = x / c;
    if (a > 100) {
      b = a + 1;
      c = b - x;
    } else
      c = x - b;
    b = c * a;
    return (b);
}
```

**(a)**

```
main:
first partition of main
            li      r3,0x5
            bl      foo
second partition of main
            addi    r12, r12, 1
            blr

foo:
single partition of foo
            li      r11, 3
            ori     r12, r3, 15
            divw    r10, r3, r11
            cmpwi   r10, 100
            bgt     L1
            sub     r11, r3, r12
            b       L2
L1:  addi   r12, r10, 1
            sub     r11, r12, r3
L2:  mullw  r12, r11, r10
            blr
```

**(b)**
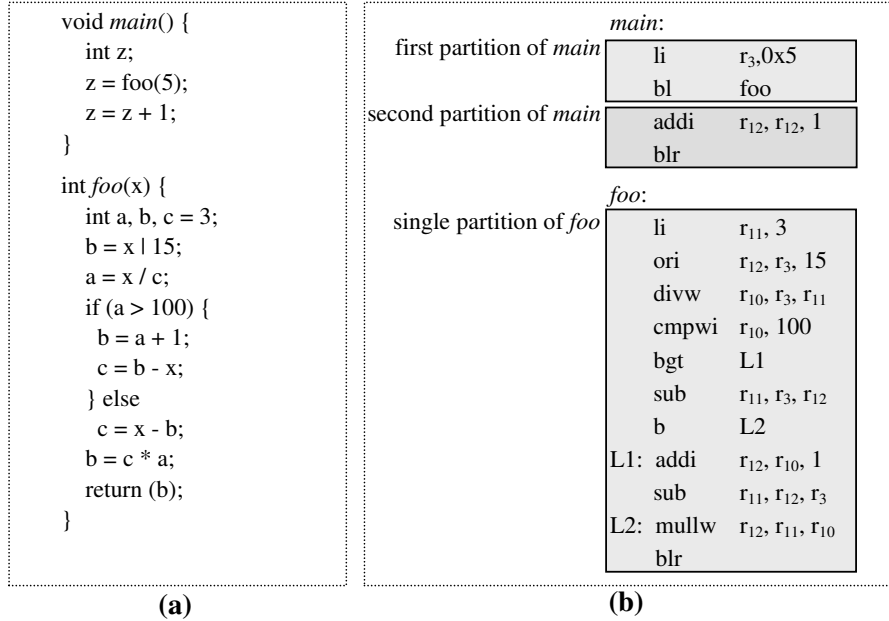
**Figure 5:** (a) An example program $T$ in C. (b) Assembly of $T$ showing the PPs.

instruction. In the PowerPC instruction set, "bl" (branch with link register update) and "blr" (branch to link register) instructions are examples for function call and indirect branch instructions, respectively.

Figure 5 shows an example program $T$. In Figure 5(b), the portion of *main* that starts from the first instruction of *main* and extends until the function call instruction in *main* is a PP inside *main*. Similarly, the remaining portion of *main* is another PP. On the other hand, since *foo* does not contain any function call instructions inside, the whole body of *foo* is a PP by itself.

## 3.3   Reversing an Assembly Instruction

In Section 3.1, we stated that the reverse program recovers only directly modified program state. Therefore, under this assumption, reversing an assembly instruction is equivalent to recovering the register and/or the memory value(s) being directly overwritten by that instruction. In this section, we outline how the RCG algorithm recovers a directly modified value.

After the PPs are determined, the RCG algorithm goes over every instruction within all PPs in *program order* and checks whether each instruction directly modifies a register or a memory location. If an instruction directly modifies a register or a memory location, the RCG algorithm generates a group of one or more instructions which recovers the overwritten value in that register or memory location. We call such a group of instructions a *reverse instruction group* (*RIG*).

A value destroyed by an instruction can be recovered in three ways: (i) the value can be recalculated during instruction level reverse execution, which we call the *redefine technique*; (ii) the value can be extracted from a previous use during instruction level reverse execution, which we call the *extract-from-use technique*; and (iii) the value can be saved during forward execution and then restored during instruction level reverse execution, which we call the *state saving technique*.

Figure 6 shows $T$ (from Figure 5) and $T$'s reverse $RT$. An instruction $i_x$ in $T$ and the generated RIG, $RIG_x$, for that instruction are marked with the same index $x$ in Figures 6(a) and 6(b), respectively. Note that the instructions that are shown in bold are extra instructions that are inserted to the original program for state saving; thus, these instructions do not have associated RIGs. In addition, as will be explained in the next section, control flow is reversed by control flow predicates inserted into the reverse code; therefore, the control flow instructions also do not have associated RIGs in the generated reverse program. Consequently, in Figure 6, we assigned indices neither to the control flow instructions in the original program nor to the state saving instructions inserted in the original program to enable reverse execution. Now, let us have a closer look at some generated RIGs.

**Example 1** Consider in Figure 6 the RIG for $i_8$ which subtracts $r_3$ from $r_{12}$ and writes the result into $r_{11}$. Since the only value that is being directly changed by $i_8$ is $r_{11}$, the generated RIG for $i_8$, $RIG_8$, should recover $r_{11}$. First of all, the RCG algorithm finds the value of $r_{11}$ that needs to be recovered. For this purpose, the RCG algorithm performs reaching definitions

```
            main:
              save   r3
     i₁       li     r3,0x5      (r3)
              bl     foo
     i₂       addi   r12, r12, 1 (r12)
              blr
            foo:
              save   r11
     i₃       li     r11, 3      (r11)
              save   r12
     i₄       ori    r12, r3, 15 (r12)
              save   r10
     i₅       divw   r10, r3, r11 (r10)
              cmpwi  r10, 100
              bgt    L1
     i₆       sub    r11, r3, r12 (r11)
              b      L2
     i₇ L1:   addi   r12, r10, 1 (r12)
     i₈       sub    r11, r12, r3 (r11)
     i₉ L2:   mullw  r12, r11, r10 (r12)
              blr

                  (a)
```

```
              foo_rev:
RIG₉              cmpwi  r10, 100
                 ble    L1
                 addi   r12, r10, 1
                 b      L2
              L1: ori    r12, r3, 15
              L2: cmpwi  r12, 100
                 ble    L3
RIG₈             li     r11, 3
RIG₇             ori    r12, r3, 15
                 b      L4
RIG₆          L3: li     r11, 3
RIG₅          L4: restore r10
RIG₄             restore r12
RIG₃             restore r11
                 b      L5
              main_rev:
RIG₂             subi   r12, r12, 1
                 b      foo_rev
RIG₁          L5: restore r3

                  (b)
```

**Figure 6:** (a) Assembly of $T$ shown in Figure 5(a). (b) Reverse of $T$, $RT$.

analysis within the PP under consideration. The definition that reaches the point just above $i_8$ is $r_{11} = 3$ which comes from $i_3$. Therefore, the value of $r_{11}$ to be recovered is 3. This value is used within the division operation at $i_5$ before being destroyed. However, the division operation does not allow the extraction of the destroyed value due to a possible loss of precision. Thus, the extract-from-use technique is inapplicable in this case. However, the redefine technique is applicable. Therefore, the RCG algorithm simply places the found reaching definition into $RIG_8$ which redefines the destroyed value and thus recovers it. □

Note that the regeneration of a value may require the regeneration of other values. For example, this would happen if the value of a register were to be regenerated using the value of another register that is also overwritten. Therefore, the redefine and extract-from-use techniques are usually applied recursively (see Section 4.2).

On the other hand, if there are multiple definitions reaching at a point along different control flow paths, then the definition to be recovered depends on the dynamic control flow of the program. In this case, the RCG algorithm generates instructions

that recover each statically reaching definition. Then, the RCG algorithm gates these instructions using conditional branches which determine (according to the dynamic control flow of the program) which definition should actually be recovered in a particular instruction level reverse execution instance. The following example illustrates this case.

**Example 2** Consider $RIG_9$ in Figure 6. In $RIG_9$, the conditional branch instruction chooses the correct definition of $r_{12}$ to be recovered based on the specific control flow path taken in forward execution of the program (see Section 4.2 for a detailed explanation). □

Finally, if a value can be recovered neither by the redefine technique nor by the extract-from-use technique, the RCG algorithm applies the state saving technique. Consider the following example.

**Example 3** In Figure 6, the value of $r_3$ that is being destroyed by $i_1$ comes from outside of the PP in which $i_1$ resides. Thus, this value cannot be redefined within the RIG to be generated. Moreover, this value is not used within the PP before being destroyed. Therefore, in this case, the RCG algorithm recovers $r_3$ by state saving which is performed by the inserted save instruction before $i_1$. Then, the generated RIG, $RIG_1$, simply restores $i_1$'s value from the saved record. □

From what we outlined above, we can state that the RCG algorithm tries to apply state regeneration by using the redefine and the extract-from-use techniques as much as possible. The RCG algorithm applies the redefine and extract-from-use techniques in combination to produce the smallest RIG. In case neither of these techniques are applicable, a RIG is generated by employing state restoration. Generally, state saving is applied if (i) the reaching definitions analysis cannot accurately find the value that is being destroyed as in the case of memory aliasing (see Section 5.1) or (ii) the value to be recovered comes from outside of the PP under consideration and thus the extraction of the value from a previous use within the PP is not possible.

We next explain the third and final main step of the RCG algorithm.

## 3.4   Combining the Reverse Instruction Groups

In order to keep the state consistent, the instructions that are executed in a certain order during forward execution are supposed to be reversed in the opposite order during instruction level reverse execution. This is similar to reversing a sequence of movie frames where the frames during fast backwarding are shown in the opposite order they are shown during forward play of a movie. In this section, we outline how we combine the RIGs in order to establish a control flow that is exactly opposite to the control flow of the original program.

From the control flow analysis point of view, we represent each PP by a control flow graph which we call a *partitioned control flow graph* (*PCFG*). A PCFG is no different than an ordinary control flow graph except that a PCFG is not necessarily constructed for a complete function or a whole program. Figure 7 (a) shows the PCFGs for the three PPs of $T$ defined previously in Section 3.2. As seen in the figure, each PCFG is further divided into *basic blocks* (*BB*s). Therefore, we represent a program in a hierarchical structure. PPs form the highest level of hierarchy, then come the BBs and finally come the individual instructions.

The RIGs are combined in accordance with our hierarchical representation of a program. From the lowest level of program hierarchy to the highest level, this combination process can be outlined as follows:

1. Combine the RIGs generated for a BB to construct the reverse of that BB which we designate as $RBB$ for short.

2. Combine the RBBs generated for a PP to construct the *reverse* of that PP which we designate as $RPP$ for short.

3. Combine the RPPs to construct the reverse program.

31
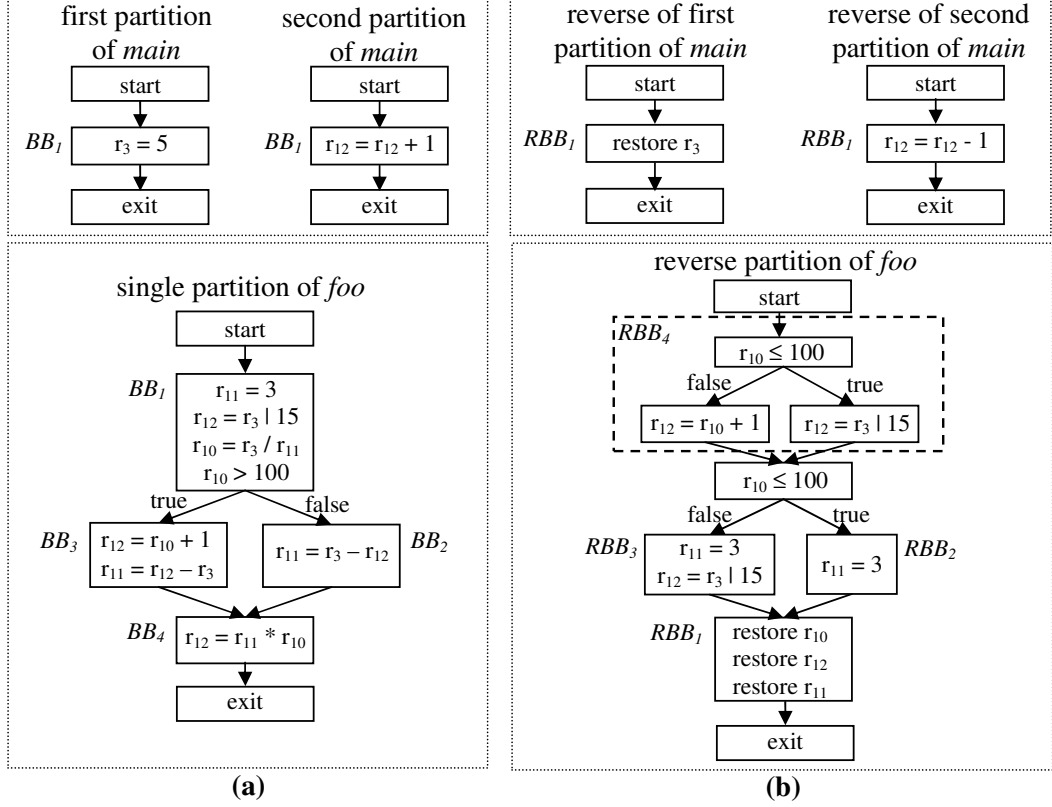
**Figure 7:** (a) PCFGs of $T$. (b) PCFGs of $RT$.

Within basic blocks of a PP, control follows a linear path. Since we want the instructions to be reversed in the opposite order they are executed in the forward direction, linear execution within basic blocks can be reversed by simply placing the RIGs in the reverse order the original instructions are placed. We call this placement order the *bottom-up placement order*. For instance, as seen in Figure 7, the RIG generated for the first instruction of $foo$ in $BB_1$ is placed at the end of the corresponding RBB, $RBB_1$; the RIG generated for the second instruction is placed on top of the previously generated RIG and so on.

We next consider the second level of hierarchy above where RBBs are to be combined into RPPs. In this case, we insert conditional branch instructions into the reverse code to provide a link between non-consecutive RBBs. Much like a railroad

track changer, conditional branch instruction(s) inserted after an RBB select the correct path to be taken following that RBB. This selection is automatically performed by the predicates of the inserted conditional branch instructions. For instance, after $RBB_4$ in Figure 7, a conditional branch is inserted which determines which RBB ($RBB_2$ or $RBB_3$) to execute next.

Finally, at the highest level of hierarchy, we combine the RPPs by using branch instructions, which is similar to the combination of the RBBs. However, unlike the branch instructions used to combine the RBBs, the target addresses of the branch instructions used to combine the RPPs might be statically determined (in case a PP is immediately reachable from a unique source address) or might also be dynamically determined (in case a PP is immediately reachable from multiple source addresses). For instance, for $T$, we statically know that $foo$ can immediately be reached from the first PP of $main$ only. Similarly, the second PP of $main$ can immediately be reached from $foo$ only. Therefore, the reverse of $foo$ is linked to the reverse of the first PP of $main$ by using a branch instruction of which target address is hard-coded. Similarly, the reverse of the second PP of $main$ is linked to the reverse of $foo$ by a branch with a hard-coded target address (see Figure 6(b)). On the other hand, the technique we use for dynamically determining the branch targets is fairly detailed; therefore, we leave the rest of this discussion to Section 4.3.

## 3.5   Summary

So far, we have introduced the basics of the three RCG steps. In the following chapter, we will present each RCG step in more detail.

# CHAPTER IV

# IMPLEMENTATION OF THE RCG ALGORITHM

This chapter explains the implementation of the three RCG steps that were briefly mentioned in the previous chapter. The first step is program partitioning which divides the subject program to smaller program regions for the ease of analysis. The second step is the generation of RIGs for the instructions of the subject program. Finally, the last step is the combination of RIGs to build a complete reverse program.

## *4.1   RCG Step 1: Program Partitioning*

In Section 3.2, we noted that the RCG algorithm divides the program under consideration into smaller program regions. In this way, the RCG algorithm can carry out control flow analysis easily. In this section, we explain the details about how we generate PPs given a program at the assembly instruction level.

To extract a PP from a program, the RCG algorithm builds a *partitioned control flow graph, PCFG*=($N$,$E$,**start**,**exit**). A PCFG is built for every PP in a 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 the PCFG, respectively. Each node in a PCFG represents a basic block (BB). Since most modern processors support only two-way branches, we assume that 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., 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 1** *Construct_PCFG()*: Construct Program Partitions

---

**Input**: A program $T$
**Output**: The PCFGs for the program partitions in $T$
**begin**
  1  $i = 0$
  2  **repeat**
  3    $PCFG_i = \phi$ /*initialize $PCFG_i$ to be empty*/
  4    $PCFG_i$ += **start** block
  5    **repeat**
  6      $\alpha$ = *Read_the_next_instruction()*
  7      **if** end of current BB is reached **then**
  8        Add current BB to $PCFG_i$
  9      **end if**
10    **until** ($\alpha$ = "function call") or ($\alpha$ = "indirect branch")
11    $PCFG_i$ += **exit** block
12    $i = i+1$
13  **until** end of the program is reached
**end**

---

Listing 1 shows the pseudo code for program partitioning. *Construct_PCFG()* builds a PCFG for each PP in the program under consideration by reading the instructions of the program in a loop (lines 5 to 10 of Listing 1). *Construct_PCFG()* starts the construction of a PCFG by inserting a **start** block at the beginning of that PCFG (line 4). Then, in the loop, *Construct_PCFG()* adds BBs to the PCFG until a function call instruction (e.g., the PowerPC "bl" – branch with link register update – instruction) or an indirect branch instruction (e.g., the PowerPC "blr" – branch to link register – instruction) is encountered in the program being analyzed. When *Construct_PCFG()* encounters a function call or an indirect branch instruction, *Construct_PCFG()* ends the construction of the PCFG by adding an **exit** block to the end of the PCFG (line 11). The instruction just after the function call or the indirect branch instruction, on the other hand, starts a new PP and thus a new PCFG.

After BBs in a program are distributed into individual PCFGs for each PP in the program, there may still be a control flow edge $e_{ij}$ from $BB_i$ in $PCFG_m$ to $BB_j$ in $PCFG_n$. We call this kind of edge an *interpartitional edge*. Since there exists a path between $BB_i$ and $BB_j$ along $e_{ij}$, the definitions made in or before $PCFG_m$ may reach $BB_j$ through $e_{ij}$. If we ignore $e_{ij}$, reaching definitions analysis performed

within $PCFG_n$ cannot determine such definitions that reach $PCFG_n$ through $e_{ij}$. Thus, the reverse code generated by reaching definitions analysis within $PCFG_n$ may be incorrect. Therefore, to be able to carry out a correct reaching definitions analysis locally, we need to find a way to represent interpartitional edges inside PCFGs. This representation is achieved as follows.

An interpartitional edge coming to $BB_j$ in $PCFG_n$ is represented by an edge from the **start** block of $PCFG_n$ to $BB_j$ in $PCFG_n$. Similarly, an interpartitional edge going out from $BB_i$ in $PCFG_m$ is represented by an edge from $BB_i$ to the **exit** block of $PCFG_m$ in $PCFG_m$. In other words, the **start** block of a PP represents all the BBs that transfer the control to that PP, and the **exit** block of a PP represents all the BBs to which control is transferred from that PP.

Note that, however, the above representation of interpartitional edges inside PPs does not imply that the program input to the RCG algorithm is modified. The branches corresponding to the interpartitional edges in the original program (e.g., a direct unconditional or conditional branch from one PP to another PP) are kept intact. These branches are represented by a separate *call graph* which will be explained in Section 4.3.3.1. Note that such a direct unconditional branch could, for example, correspond to a "goto" in the original source code.

The following example covers PP generation but does not address control flow among PPs (e.g., from $PP_i$ to $PP_j$ or $PP_k$).

**Example 4** Figure 8(a) shows a sample program portion $T$. The original control flow graph and the partitioned control flow graphs for $T$ are also shown in Figures 8(b) and 8(c), respectively. Note that in Figure 8(a), the symbolic information $foo_1$ and $foo_2$ are shown for explanation purposes (as stated in Section 3.1.1, our actual input is only the assembly code, nothing else). In a compiled code, $foo_1$ and $foo_2$ are replaced by actual addresses. In Figure 8, the solid edges indicate the intrapartitional edges while the dotted edges indicate the interpartitional
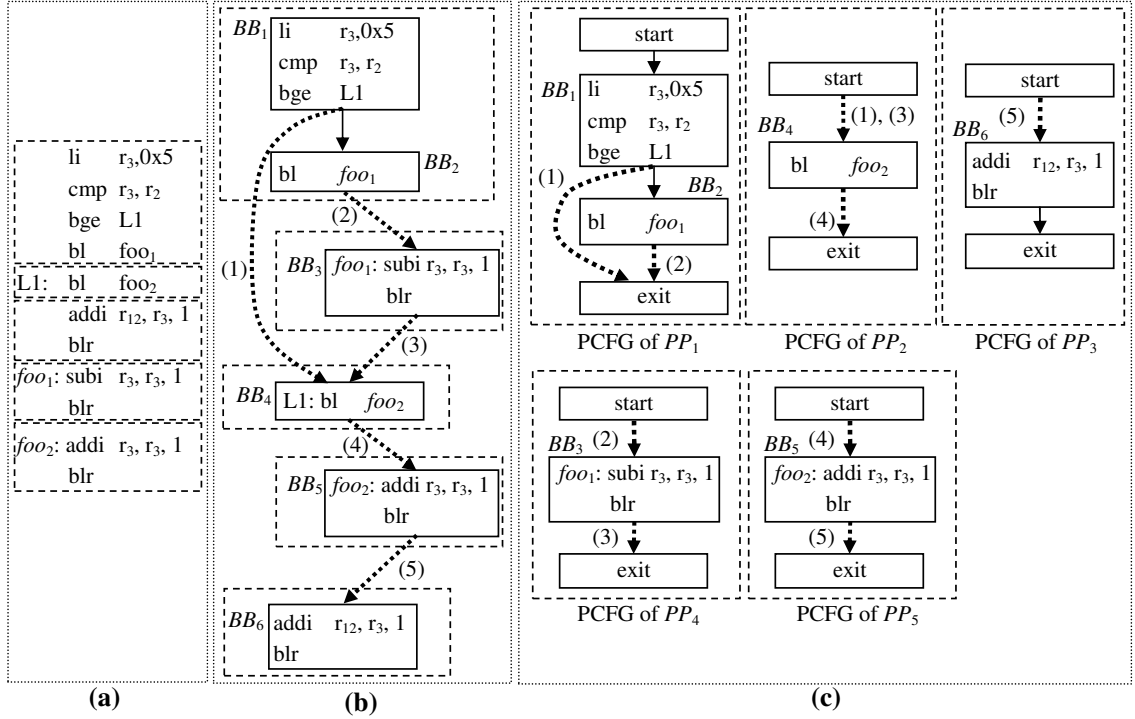
**Figure 8:** (a) A sample program portion $T$. (b) Corresponding control flow graph. (c) Corresponding PCFGs.

edges. Also, in Figure 8, the solid rectangles indicate BB boundaries while the dotted rectangles indicated the PP boundaries.

In Figure 8(c), the original control flow graph shown in Figure 8(b) is divided into five PPs (i.e., $PP_1$, $PP_2$, $PP_3$, $PP_4$ and $PP_5$) at points of "bl" (branch with link register update) and "blr" (branch to link register) PowerPC instructions (line 10 of Listing 1). Each partitioned control flow graph is delimited with a **start** and an **exit** block (lines 4 and 11 of Listing 1). Note that the interpartitional edge indicated by (1) that leaves $BB_1$ in the original control flow graph is represented by an edge from $BB_1$ to the **exit** block in the PCFG of $PP_1$. Similarly, the interpartitional edge indicated by (2) that leaves $BB_2$ in the original control flow graph is represented by an edge from $BB_2$ to the **exit** block in the PCFG of $PP_1$. We will address later in Section 4.3.3.1 and Example 10 how an interpartitional edge in the original control flow graph

(e.g., the edge from $BB_2$ in Figure 8(b) to $foo_1$) is represented in a call graph to perform an interpartitional control flow analysis (e.g., in order to be able to direct the control from $foo_1$ back to $BB_2$ during reverse execution.)

The representation of interpartitional edges inside the PCFGs can be seen in the rest of the PPs of $T$ as well. For instance, the interpartitional edges indicated by (1) and (3) that come to $BB_4$ in the original control flow graph are represented by an edge from the **start** block to $BB_4$ in the PCFG of $PP_2$. Similarly, the interpartitional edge indicated by (5) that comes to $BB_6$ in the original control flow graph is represented by an edge from the **start** block to $BB_6$ in the PCFG of $PP_3$. □

## 4.2   RCG Step 2: RIG Generation

A RIG reverses the effect of an instruction that directly modifies a register or a memory location. This section presents a detailed description of RIG generation.

Suppose that a definition $\delta_{destroy}$ destroys the value $D$ of a variable $V$ (a directly modified register or memory location) at a program point as shown in Figure 9. Let us name the program point just before $\delta_{destroy}$ as **P**.

Each statically reaching definition $\delta_i$ of $V$ at point **P** might correspond to the instance where $D$ is actually assigned to $V$ (Figure 9). 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

$D \varepsilon \{D_1, D_2, D_3\}$
Re-executing $\delta_1$ recovers $D$ for path 1
Re-executing $\delta_2$ recovers $D$ for paths 2 and 3
Re-executing $\delta_3$ recovers $D$ for path 4
Extracting $V$ out of $\mu$ recovers $D$ for paths 3 and 4
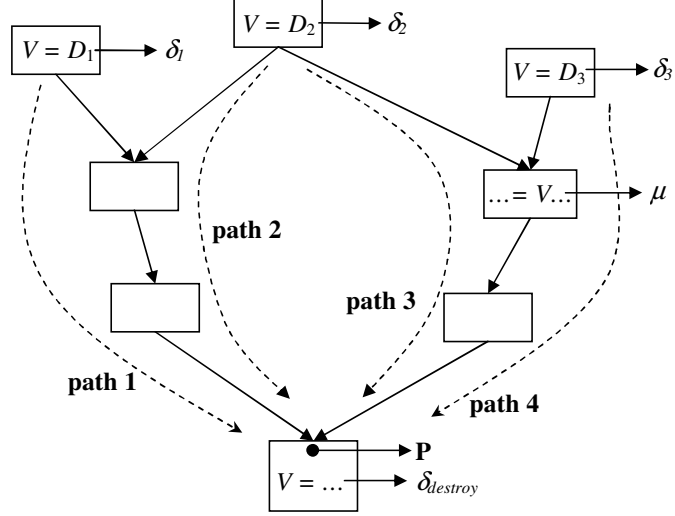
**Figure 9:** Recovering a destroyed variable $V$.

to Figure 9, we can generate a set that recovers $\delta_1$. This set indeed recovers $D$ if **path 1** is dynamically taken. Similarly, we can generate another set that recovers $\delta_2$. 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. 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 $\zeta$ can be generated to recover at least one definition of $V$ reaching $\mathbf{P}$. As briefly mentioned in Section 3.3, there are three techniques that are followed to generate a $\zeta$: the *redefine technique*, the *extract-from-use* technique and the *state saving* technique.

### 4.2.1 The Redefine Technique

The redefine technique places into $\zeta$ the instruction $\alpha_i$ that computes $D_i$ at the definition $\delta_i$ statically reaching point $\mathbf{P}$ (Figure 9). If any one of the variables, say $x_i$, that is used for computing $D_i$ is also destroyed, then the instruction that recovers $x_i$ must be inserted before $\alpha_i$ in $\zeta$; this must be applied recursively for all other modified variables in the dependency chain. For instance, assume that a register $r_i$ depends on another register $r_j$, $r_j$ depends on $r_k$ and $r_k$ does not depend on any other variable in the program. Therefore, there is a dependency chain from $r_i$ to $r_k$. Also, assume that both $r_i$ and $r_j$ are destroyed, while $r_k$ is available. Then, using the redefine technique recursively, the RCG algorithm first recovers $r_j$ using $r_k$ and then recovers $r_i$ using $r_j$.

The redefine technique can potentially recover only one definition $\delta_i$ of $V$ reaching $\mathbf{P}$: namely, the definition $\delta_i$ which is redefined. Therefore, if a variable has more than one statically reaching definition (e.g., $\delta_i$ and $\delta_j$) at a point in a program, the redefine technique has to be applied, if applicable, to each statically reaching definition (e.g., both $\delta_i$ and $\delta_j$) separately. Note that, however, the external value of an input variable (e.g., a global variable or an input argument) of a PP is certainly not defined within the PP but comes from outside of the PP. Therefore, the external values of variables of a PP cannot be recovered by the redefine technique.

The following example illustrates how the redefine technique works.

**Example 5** *The redefine technique*: Consider the instruction that overwrites the value of register $r_{12}$ in $BB_4$ in Figure 10 (we need the overwritten value of $r_{12}$ because the overwritten value is used both in $BB_2$ and $BB_3$). Let us name this instruction as $\alpha$ (i.e., $\alpha =$ "$r_{12} = r_{11} * r_{10}$")

and the analysis points just before $\alpha$ as $\mathbf{P}$ and just after $\alpha$ as $\mathbf{P'}$. 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, $r_{12}$ can be recovered on one path by executing the set $\zeta_1 = \{(r_{12} = r_{10} + 1)\}$, and $r_{12}$ can be recovered on the other path by executing the set $\zeta_2 = \{(r_{12} = r_3 \mid 15)\}$. $\square$
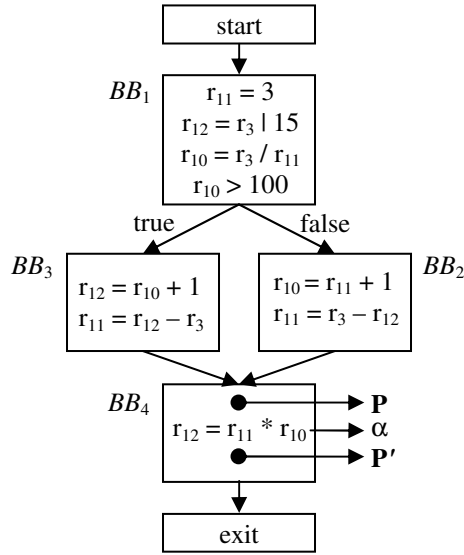


**Figure 10:** An example PCFG of a program partition $PP_x$.

### 4.2.2 The Extract-from-use Technique

The extract-from-use technique places into $\zeta$ an instruction $\beta$ that extracts the destroyed value of $V$ out of a use $\mu$ (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 9). However, again, if any other variable $x_i$ in $\beta$ that is used for extracting $V$ is also destroyed, then an instruction that recovers $x_i$ must be inserted before $\beta$ in $\zeta$; 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 (e.g., both $\delta_2$ and $\delta_3$ in Figure 9) of $V$ reaching $\mathbf{P}$. Moreover, since the external value of an input variable of a PP may be used within the PP, the input values to a PP may be recoverable by use of 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 $\mu$ on a path to $\delta_{destroy}$, and, even if a use is available, $\mu$'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", "integer multiply" and "shift" allow extraction of values provided that the information that will allow such an extraction is not lost due to an overflow/underflow or a shift-out during these operations (note that as mentioned in Section 3.1, we assume the programmer notifies RCG if such overflow/underflow or shift-out is possible; if overflow/underflow or shift-out is possible, RCG disables any use whatsoever of the extract-from-use technique). On the other hand, operations such as "integer divide" and floating point calculations do not typically allow extraction of values due to a possible loss of precision on the result.

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

**Example 6** *The extract-from-use technique*: Consider again the instruction named $\alpha$ in Figure 10. 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

42

obtained by executing the set $\zeta_3 = \{(r_{12} = r_{11} + r_3)\}$; 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 $\zeta_4 = \{(r_{12} = r_3 - r_{11})\}$. $\square$

### 4.2.3 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 9). Even worse, as in the case of memory aliasing which will be described in Section 5.1, we may not be able to find the statically reaching definitions of $V$ at all. In such circumstances, the RCG algorithm resorts to the state saving technique.

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 11) 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 that moves the memory pointer to the next value to be restored and restores the saved value from memory.
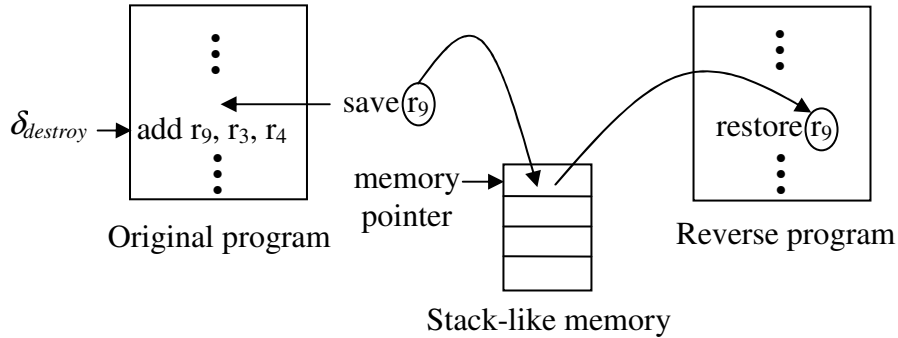


**Figure 11:** A diagram illustrating the state saving method of the RCG algorithm.

A push-/pop-like instruction refers to an instruction that 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, the PowerPC 860 provides store-update and load-update instructions that 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 that automatically increment/decrement a memory pointer, save and restore operations are handled by using ordinary store and load instructions with explicit increment/decrement of a dedicated memory pointer.

### 4.2.4 An Example of RIG Generation

In the previous three subsections, we explained the three methods (redefine, extract-from-use and state saving) we use to generate a set that 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 12.

**Example 7** *RIG generation*: In Examples 5 and 6, we gave four different sets $\zeta_1$, $\zeta_2$, $\zeta_3$ and $\zeta_4$ each of which recover the value of $r_{12}$ along a particular path to point $\mathbf{P}$ in Figure 12. Let us now pick some of these sets to cover all the paths to point $\mathbf{P}$ and combine the selected sets to generate a complete RIG for recovering the value of $r_{12}$. Let us pick $\zeta_1 = \{(r_{12} = r_{10} + 1)\}$ to recover $r_{12}$ along the left path to point $\mathbf{P}$ and $\zeta_4 = \{(r_{12} = r_3 - r_{11})\}$ to recover $r_{12}$ along the right path to point $\mathbf{P}$ in Figure 12 (note that the RCG algorithm always chooses minimum size sets to produce a smallest possible RIG). Since all paths to point $\mathbf{P}$ are covered, $\zeta_1$ and $\zeta_4$ are enough to generate a RIG.
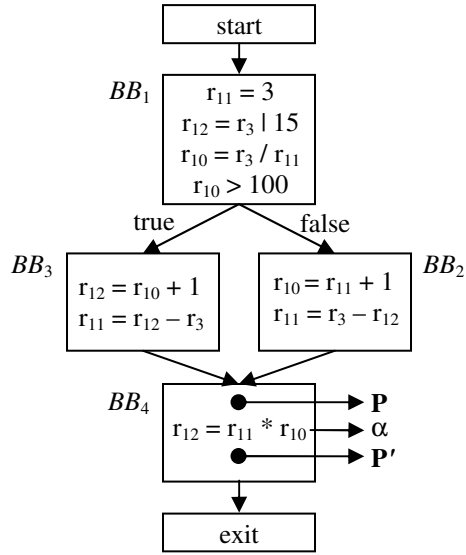
**Figure 12:** PCFG from Figure 10.

We should now combine $\zeta_1$ and $\zeta_4$ by using a conditional branch instruction that determines along which path **P** is reached. The predicate of this conditional branch instruction is $r_{10} > 100$. However, we cannot use this predicate directly because the value of $r_{10}$ is destroyed in $BB_2$. 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 $r_{10}$ in order not to destroy the values of $r_{11}$ and $r_{10}$ at point **P**. Therefore, a RIG for recovering $r_{12}$ in PowerPC assembly can be generated as follows:

$$
\begin{aligned}
&li \quad\quad r_t,\ 3 \\
&divw \quad r_t,\ r_3,\ r_t \\
&cmpwi \quad r_t,\ 100 \\
&bgt \quad\quad L1 \\
&sub \quad\quad r_{12},\ r_3,\ r_{11}
\end{aligned}
$$

$$b \qquad L2$$

$$L1: \quad addi \quad r_{12}, r_{10}, 1$$

$$L2: \quad ...$$

□

Listing 2 shows the pseudo code for the generation of a RIG corresponding to an instruction $\alpha$. To find the minimum sized RIG, *Gen_RIG()* first finds all reaching definitions $\delta$ of the register/memory location $t$ modified by $\alpha$ (line 3). Then, *Gen_RIG()* creates an empty set $\zeta_\delta$ (line 4) and sends $\zeta_\delta$ together with the definitions to be recovered to function *Recover()*. *Recover()* applies the extract-from-use technique and the redefine technique recursively to recover $\delta$ and fills in the set passed by *Gen_RIG()* (the pseudo code for *Recover()* will be shown in Section 5.1.3). Note that an instruction may directly modify more than one physical location (e.g., the store-update instruction "stwu $r_2$, $4(r_1)$" on the PowerPC 860 modifies the memory location at address $r_1 + 4$ and then increments $r_1$ by four, modifying both a memory location and a register location with a single instruction). Therefore, as the "for all" statement at line 2 indicates, *Gen_RIG()* repeats the main loop between line 2 and line 11 adding to the final RIG the instructions that recover each modified location (line 10).

To recover a value, *Recover()* searches many different uses (with reversible operators) and/or definitions on different paths, where each use/definition covers a set of one or more paths. If $\zeta_\delta$ returned from *Recover()* has a cost of infinity (line 7), this means that neither the extract-from-use technique nor the redefine technique can recover $t$; thus, *Gen_RIG()* applies the state saving technique to recover $t$ (line 8).

**Listing 2** *Gen_RIG()*: Generate a RIG

---

**Input**: An instruction $\alpha$
**Output**: A RIG, $RIG_\alpha$, for $\alpha$
**begin**
  1  $RIG_\alpha = \phi$
  2  **for all** $t =$ a register/memory location directly modified by $\alpha$ **do**
  3     $\delta = Find\_Reaching\_Def(t, \alpha)$
  4     $\zeta_\delta = \phi$
  5     $\zeta_\delta.\text{Cost} = \infty$
  6     $Recover(\delta, \zeta_\delta)$
  7     **if** $\zeta_\delta.\text{Cost} == \infty$ **then**
  8       $\zeta_\delta = \text{State\_save}(t)$
  9     **end if**
10     $RIG_\alpha \mathrel{+}= \zeta_\delta$
11 **end for**
**end**

---

# 4.3  RCG Step 3: Combining the RIGs

The last step to build a reverse program is to combine the RIGs together. As mentioned in Section 3.3, RIG combination is a hierarchical process. In the lowest level, RIGs are combined into RBBs. Then, in the next higher level, RBBs are combined into RPPs. Finally, at the highest level, RPPs are merged to form the reverse program. The following three subsections present each level of hierarchy in detail.

### 4.3.1  Constructing the RBBs

The first step to combine the RIGs is to build the reverse of each basic block in the original program. The instructions within a BB complete in lexical order; thus, to keep the state consistent during instruction level reverse execution, the RIGs should execute in an order exactly opposite of the order of execution of the instructions in the original program's BB. Therefore, placing the RIGs in the order opposite to the lexical order of a BB is sufficient to generate the reverse of that BB. In other words, if a basic block $BB_i$ in the PP 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 7), 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 PP.
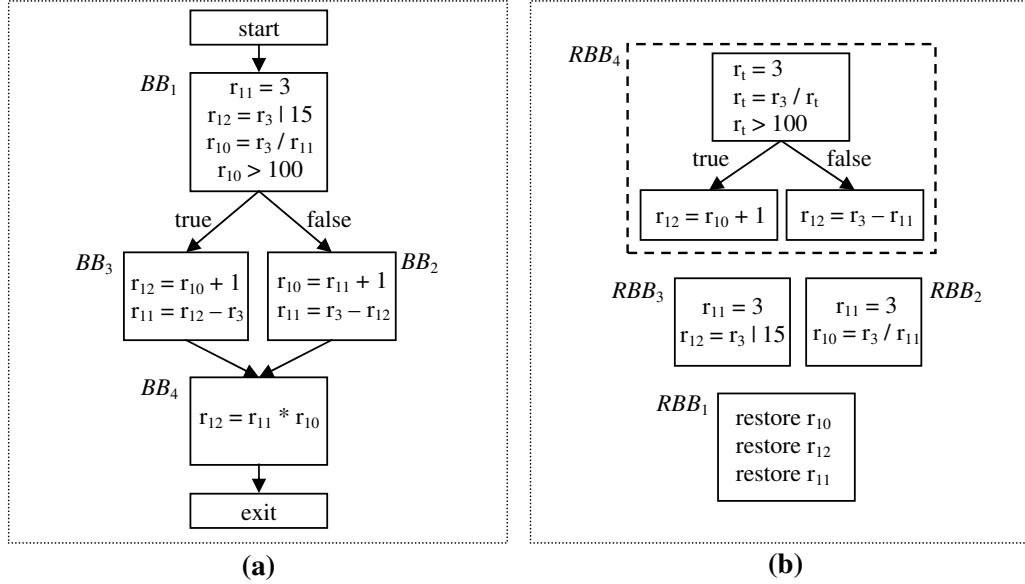


**Figure 13:** (a) PCFG of $PP_x$ from Figure 10. (b) RBBs of reverse program partition $RPP_x$.

**Example 8** *Constructing the RBBs*: Figure 13(a) shows the PCFG of $PP_x$ previously shown in Figure 10, and Figure 13(b) shows the RBBs generated for the reverse of $PP$, $RPP_x$. The RCG algorithm generates the reverse of each BB in $PP_x$ by combining the generated RIGs in bottom-up placement order in $RPP_x$. While the reverse of $BB_1$, $BB_2$ and $BB_3$ (namely, $RBB_1$, $RBB_2$ and $RBB_3$) are constructed each as a single BB, the reverse of $BB_4$, $RBB_4$, consists of three separate BBs. $RBB_4$ is separated into three BBs because the reverse of the instruction "$r_{12} = r_{11}$ * $r_{10}$" in $BB_4$ consists of multiple instructions two of which are branches (as given in the assembly listing in Example 7). □
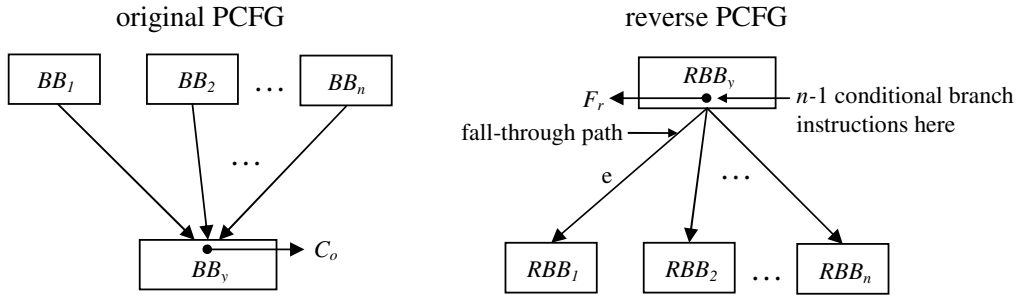
**Figure 14:** Combination of RBBs at a fork point.

### 4.3.2 Constructing the RPPs

To generate the reverse of a PP, the RBBs generated for that PP should be combined in an appropriate way. Once again, this combination should satisfy our argument that the RIGs should execute in the order opposite to the execution order of the instructions in the original program. Since edges in a PCFG designate the control flow between the BBs, we reverse the control flow simply by combining the RBBs via the inverted versions of the edges in the PCFG of the original PP.

Inverting the edges implies two facts. First, the RBBs are placed in an order opposite to the order of BBs in a program. This is same as the bottom-up placement order of RIGs within an RBB. Second, a confluence point of incoming edges in a PCFG typically becomes a fork point of outgoing edges in the reverse version of that PCFG, and vice versa. Consequently, one or more conditional branch instructions are needed at the generated fork points in the reverse program.

Suppose that a confluence point $C_o$ in a PCFG becomes a fork point $F_r$ in the reverse PCFG as seen in Figure 14. Depending on the number of incoming edges to $C_o$ (or outgoing edges from $F_r$), the RCG algorithm inserts at $F_r$ one or more conditional branch instructions that decide on which edge to take at $F_r$ during instruction level reverse execution (Figure 14).

49
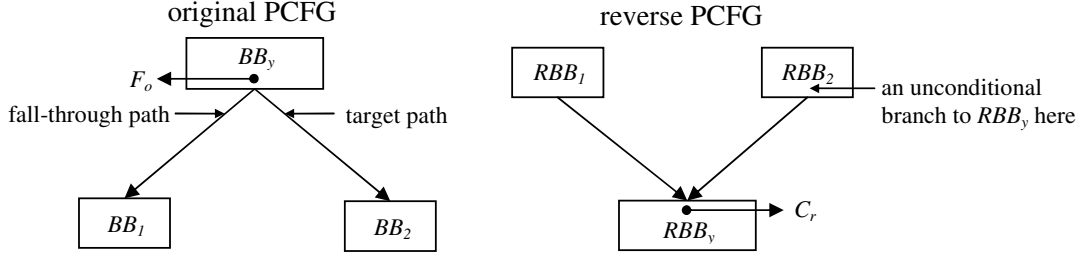
**Figure 15:** Combination of RBBs at a confluence point.

Due to linear orientation of code in memory, typically 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 $e$ (Figure 14). Since it is inefficient to generate a conditional branch whose target address is the next address in memory, the RCG algorithm omits the generation of a conditional branch instruction for $e$. A conditional branch instruction is generated for all other edges leaving $F_r$. Therefore, due to our prior assumption that the target processor architecture supports only two-way branches (see Section 4.1), the number of conditional branches to be inserted at a fork point with $n$ targets is $n-1$.

The decision on which edge to take at $F_r$ is automatically performed during instruction level reverse execution by the predicates of the conditional branch instructions inserted at $F_r$. This approach is similar to the use of predicates for choosing the correct set to execute inside a RIG (see Section 4.2). Since $F_r$ corresponds to $C_o$, the predicates to be chosen at $F_r$ are essentially the same as the predicates that determine along which edge $C_o$ is dynamically reached during forward execution.

On the other hand, suppose that a fork point $F_o$ in a PCFG becomes a confluence point $C_r$ in the reverse PCFG (recall from Section 4.1 that a fork point in a PCFG can have at most two outgoing edges at the assembly level). This scenario is depicted in Figure 15. In this case, it is necessary to establish a link between $C_r$ and each RBB that is the source of one of the joining edges at $C_r$ ($RBB_1$ and $RBB_2$ in Figure 15).

Again, due to linear orientation of code in memory, the RBB that corresponds to the BB on the fall-through path of $F_o$ (e.g., $RBB_1$ in Figure 15) will always immediately precede $C_r$ in the reverse code. Hence, a link between $RBB_1$ and $C_r$ is already established. Therefore, the remaining part is to provide the link between $C_r$ and $RBB_2$ that corresponds to the BB on the target path of $F_o$. This link is established by inserting an unconditional branch at the end of $RBB_2$ as shown in Figure 15.

The following example illustrates how the RBBs are combined to generate an RPP.

**Example 9** *Constructing the RPPs*: Figure 16 shows the PCFG of $PP_x$ and the PCFG of the corresponding RPP, $RPP_x$. Also seen in the figure are the assembly listings of the instrumented $PP_x$ (i.e., instrumented with state saving instructions) and $RPP_x$. As seen by the assembly listings, the RBBs are placed in bottom-up placement order in memory. Since the RBBs are combined with the inverted versions of the edges in the PCFG of $PP_x$, the confluence point designated as $C_o$ in the PCFG of $PP_x$ becomes a fork point designated as $F_r$ in the PCFG of $RPP_x$, and the fork point designated as $F_o$ in the PCFG of $PP_x$ becomes a confluence point designated as $C_r$ in the PCFG of $RPP_x$. Consequently, a conditional branch instruction is inserted at point $F_r$ and an unconditional branch instruction is inserted at the head of one of the joining edges at $C_r$.

The predicate of the conditional branch inserted at point $F_r$ in $RPP_x$ is $r_{10} > 100$. Since $r_{10}$ is not available at point $C_o$ which corresponds to point $F_r$ in $RPP_x$, this predicate is first recovered just like it is recovered in $RBB_4$ (see Example 7). Also, as seen in Figure 16, instead of using $r_{10} > 100$, the RCG algorithm uses the complementary predicate $r_{10} \leq 100$ at point $F_r$. This can be explained as follows: In $PP_x$, $BB_2$ immediately follows the conditional branch at point $F_o$. However, due to the bottom-up placement order of RBBs, the situation is opposite in the reverse code. Namely, instead of $RBB_2$, $RBB_3$ immediately follows the conditional

## PCFG of $PP_x$

start

$BB_1$
$r_{11} = 3$
$r_{12} = r_3 \mid 15$
$r_{10} = r_3 / r_{11}$
$r_{10} > 100$ → $F_o$

target path          fall-through path

$BB_3$
$r_{12} = r_{10} + 1$
$r_{11} = r_{12} - r_3$

$r_{10} = r_{11} + 1$
$r_{11} = r_3 - r_{12}$          $BB_2$

**conditional branch**

$BB_4$
$r_{12} = r_{11} * r_{10}$ → $C_o$

exit

**unconditional branch**

## PCFG of $RPP_x$

start

$RBB_4$
$r_t = 3$
$r_t = r_3 / r_t$
$r_t > 100$
true          false
$r_{12} = r_{10} + 1$          $r_{12} = r_3 - r_{11}$

$r_t = 3$
$r_t = r_3 / r_t$
$r_t \leq 100$ → $F_r$

fall-through path          target path

$RBB_3$
$r_{11} = 3$
$r_{12} = r_3 \mid 15$

$r_{11} = 3$
$r_{10} = r_3 / r_{11}$          $RBB_2$

$RBB_1$
restore $r_{10}$ → $C_r$
restore $r_{12}$
restore $r_{11}$

exit

## instrumented $PP_x$

```
        li      r₉, 0x0
        stwu    r₁₁, 4(r₉)
        li      r₁₁, 3
        stwu    r₁₂, 4(r₉)
        ori     r₁₂, r₃, 15      } BB₁
        stwu    r₁₀, 4(r₉)
        divw    r₁₀, r₃, r₁₁
        cmpwi   r₁₀, 100
        bgt     L1
        sub     r₁₁, r₃, r₁₂     } BB₂
        b       L2
L1:     addi    r₁₂, r₁₀, 1      } BB₃
        sub     r₁₁, r₁₂, r₃
L2:     mullw   r₁₂, r₁₁, r₁₀    } BB₄
        blr
```

## $RPP_x$

```
        li      r_t, 3
        divw    r_t, r₃, r_t
        cmpwi   r_t, 100
        bgt     L1
        sub     r₁₂, r₃, r₁₁     } RBB₄
        b       L2
L1:     addi    r₁₂, r₁₀, 1
L2:     li      r_t, 3
        divw    r_t, r₃, r_t
        cmpwi   r_t, 100
conditional → ble     L3
branch        li      r₁₁, 3      } RBB₃
              ori     r₁₂, r₃, 15
unconditional → b       L4
branch      L3: li      r₁₁, 3      } RBB₂
                divw    r₁₀, r₃, r₁₁
            L4: lwzu    r₁₀, -4(r₉)
                lwzu    r₁₂, -4(r₉)  } RBB₁
                lwzu    r₁₁, -4(r₉)
```

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 16:** A diagram illustrating the combination of the RBBs.

branch inserted at point $F_r$ (see assembly listing of $RPP_x$). Therefore, the predicate is inverted in the reverse code.

Note that an unconditional branch instruction is placed only at the end of $RBB_3$ that corresponds to the BB on the fall-through path of the fork in $PP_x$ (the other RBB, $RBB_2$, simply precedes $RBB_1$ in address space). □

### 4.3.3   Combining the RPPs

After generating an RPP, the RCG algorithm combines that RPP with the other RPPs that have already been generated (the overall flow was explained in Chapter 3, Figure 3). In order to achieve this, the RCG algorithm must know the control flow information between the PPs in the program under consideration.

#### 4.3.3.1   Determining the Control Flow Information Between PPs

Since PCFG construction is performed for each PP separately, each PCFG designates the control flow within a particular PP only. In other words, a PCFG does not contain any edges that show the flow of control between the PPs. Therefore, the control flow information between the PPs is determined by another graph, $G=(N,E,s,t)$, which is a *call graph* ($CG$).

In $G=(N,E,s,t)$, the set $N$ is the set of nodes designating the PPs in a program and the set $E$ is the set of edges designating the flow of control between those PPs. The notations $s$ and $t$ designate the unique entry and exit nodes of a CG. Note that an indirect branch whose target PP is statically unknown may potentially invoke any PP. Therefore, if a PP, $PP_i$, makes an indirect call whose target PP is statically unknown, the RCG algorithm inserts an edge from $PP_i$ to every other PP.

To learn from which address(es) a PP can be immediately reached and thus to be able to move the control backwards to a source address, the RCG algorithm annotates

**Listing 3** *Grow_CG()*: The CG construction algorithm

---

**Input**: A program partition $PP_j$ 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 $PP_j$
  2  **for all** $PP_k$ immediately reached from $PP_j$ **do**
  3    **if** $(n_k = \text{node\_of}(PP_k)) \neq \text{NULL}$ **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**

---

an edge $e_{ij} \in E$ from a PP, $PP_i$, to another PP, $PP_j$, with the address of the instruction in $PP_i$ that transfers control from $PP_i$ to $PP_j$.

Listing 3 shows the pseudo code for call graph construction. *Grow_CG()* adds a new node to the CG for a PP when a PCFG is built for that PP. After a new node $n_j$ is generated for a PP, $PP_j$, *Grow_CG()* checks the PPs that are immediately reachable from $PP_j$. For every PP, $PP_k$, that is immediately reachable from $PP_j$ and for which a PCFG (and thus a node in the CG) has already been generated, *Grow_CG()* adds an edge $e_{jk}$ from the node of $PP_j$ to the node of $PP_k$ and annotates $e_{jk}$ with the address of the instruction transferring control from $PP_j$ to $PP_k$ (lines 2 to 5 of Listing 3). For every other PP that is immediately reachable from $PP_j$ but for which a node has not yet been generated, *Grow_CG()* sets a pending edge (lines 6 and 7). Then, *Grow_CG()* checks whether $PP_j$ has pending incoming edges set for it. If $PP_j$ has pending incoming edges, *Grow_CG()* adds to the CG all the pending incoming edges that are set for $PP_j$ and annotates those edges appropriately (lines 10 to 15).
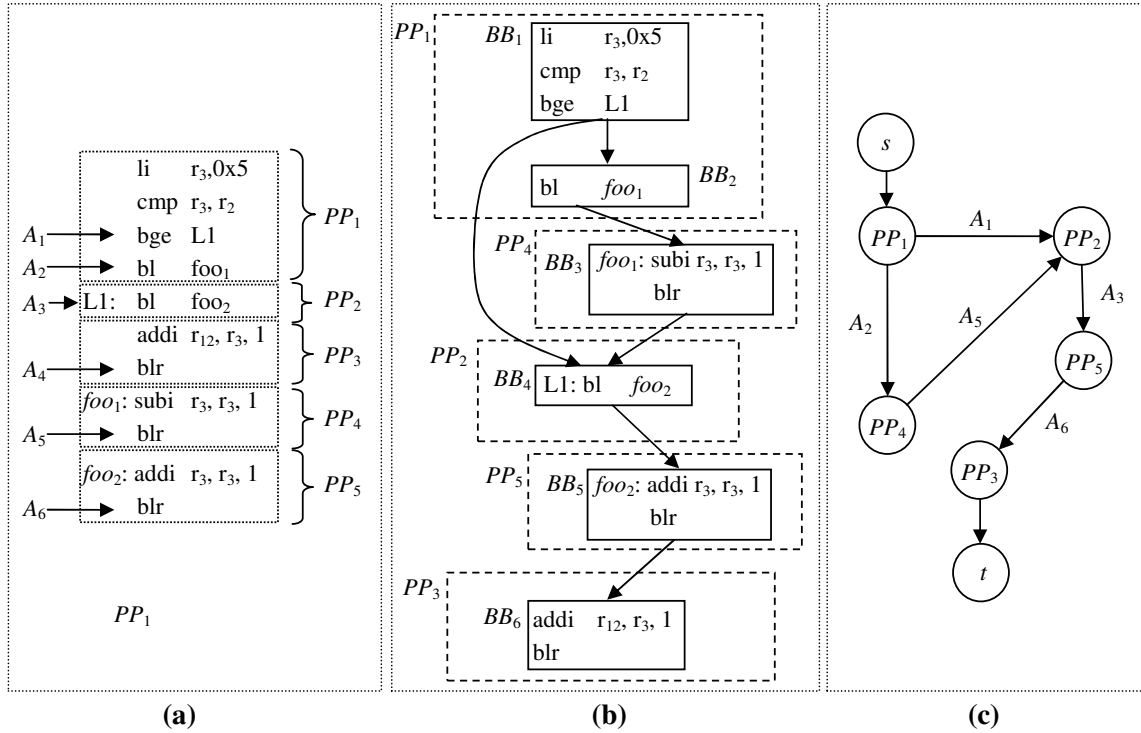
**Figure 17:** (a) The example program portion from Figure 8. (b) Corresponding control flow graph. (c) Corresponding call graph (CG).

**Example 10** *Call graph construction*: Figure 17(a) shows the sample program portion $T$ from Example 4. Figure 17(b) shows the control flow graph for $T$, and Figure 17(c) shows the CG for $T$. As previously described in Example 4, $T$ consists of five PPs: $PP_1$, $PP_2$, $PP_3$, $PP_4$ and $PP_5$. The edges in the CG show the transfer of control between the PPs and are annotated with the addresses of the instructions transferring the control between PPs. For example, at address $A_1$, control may be directed from $PP_1$ to $PP_2$ if the conditional branch "bge L1" is taken. Therefore, in the CG, there exists an edge with annotation $A_1$ from $PP_1$ to $PP_2$. As another example, at the end of $PP_4$ the "blr" (branch to link register) instruction directs the control to the address following the function call instruction "bl" in $PP_1$ which is used to call $PP_4$ from $PP_1$. The address following the "bl" instruction in $PP_1$ is the beginning address of $PP_2$; thus, there exists an edge with annotation $A_5$ (the address of "blr" instruction in $PP_4$)
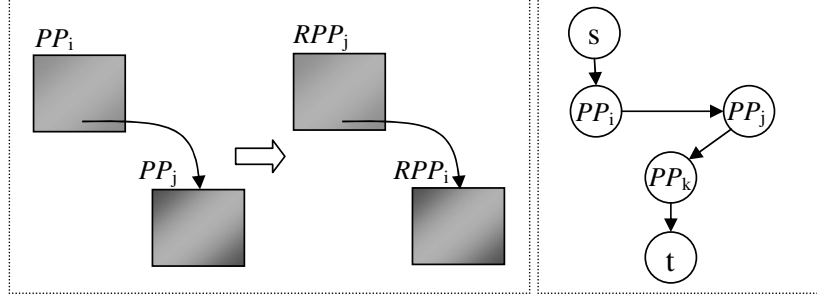
**Figure 18:** An abstract view of two PPs.

from $PP_4$ to $PP_2$ in the CG. As seen in Figure 17(c), the CG captures similar other control flow information between the PPs. □

*4.3.3.2   Using the CG to Combine the RPPs*

Equipped with the control flow information between PPs, the final step in the RCG algorithm is to invert that control flow to combine the RPPs into a complete reverse program. In this section, we describe how we use a CG to achieve this task.

Figure 18 shows an abstract view of two PPs, $PP_i$ and $PP_j$, the reverses of these PPs, $RPP_i$ and $RPP_j$, and a sample call graph. As seen in the figure, there is an edge from $PP_i$ to $PP_j$ in the CG. This means that $PP_j$ is immediately reachable from $PP_i$. If, indeed, $PP_j$ is reached from $PP_i$ during a specific execution, then after reverse executing $PP_j$, the control must be returned to $PP_i$. Therefore, in this case, we must provide a link from the reverse of $PP_j$ to the reverse of $PP_i$ in the reverse program.

If a program partition $PP_i$ is immediately reachable 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 $PP_i$ 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 $PP_i$, $RPP_i$, is executed. This is easily handled by inserting at the end of $RPP_i$ an unconditional branch instruction whose

56

target address is $RA$. However, if a program partition $PP_i$ is immediately reachable from multiple static locations (i.e., there are multiple edges coming into the node of $PP_i$ in the CG), the location from which $PP_i$ 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 $RPP_i$, the reverse of $PP_i$, cannot be obtained statically. Therefore, in such a case, the RCG algorithm applies a dynamic technique, called the *stack-tracing technique*, to find the location to which the control should be directed after executing $RPP_i$. We next describe the stack-tracing technique.

**The stack-tracing technique**

The stack-tracing technique can simply be described as saving the statically unknown return addresses of RPPs into a stack at runtime. During reverse execution, the saved addresses are popped back from the stack to provide return from an RPP. There are similar well-known forms of the stack-tracing technique in the literature that are usually applied for obtaining call traces of procedures. In the context of reverse code generation, we apply the state-tracing technique to learn in which order the PPs are visited in a specific forward execution and thus to be able to reverse that order during reverse execution.

Let us assume that a subset $\Phi_{PP}$ of the PPs in the program under consideration contains all PPs which are immediately reachable from multiple static locations. We will designate the set of the reverses of these PPs as $\Phi_{RPP}$. Thus, the remaining PPs in the program, but not in $\Phi_{PP}$, are immediately reachable from a single static location each. Also, assume that there are a total of $n$ locations from which control reaches the PPs in $\Phi_{PP}$. We will designate the addresses of these $n$ locations as $\Phi_A = \{A_1, A_2, \ldots A_n\}$. We will also designate the corresponding $n$ addresses in the reverse code as $\Phi_{RA} = \{RA_1, RA_2, \ldots RA_n\}$. Therefore, after executing the reverse of a program partition $PP_i \in \Phi_{PP}$ during instruction level reverse execution, control

**Listing 4** *Stack_Trace()*: The stack-tracing algorithm

---

**Input**: A call graph $CG$
**begin**
  1  **for all** node $PP_i \in CG$ **do**
  2    **if** $|\text{incoming\_edges}(PP_i)| > 1$ **then**
  3      **for all** program partitions $PP_j$ corresponding to parents of $PP_i$ in $CG$ **do**
  4        $RA = \text{Beginning\_Address}(\text{reverse}(PP_j))$
  5        insert 'push $RA$' at the end of $PP_j$
  6      **end for**
  7      insert 'pop $RA$' and 'branch to $RA$' at the end of $RPP_i$ (reverse of $PP_i$)
  8    **end if**
  9  **end for**
**end**

---

should be directed to an address $RA_i$ if and only if the control has reached $PP_i$ from the corresponding address $A_i$ during forward execution ($A_i \in \Phi_A$, $RA_i \in \Phi_{RA}$).

The addresses to which control should be transferred from an RPP in $\Phi_{RPP}$ during a specific reverse execution of the program under consideration can be found by saving the addresses in $\Phi_{RA}$ into a runtime stack during forward execution. In other words, whenever a transfer from an address $A_i$ in $\Phi_A$ to a program partition $PP_j$ in $\Phi_{PP}$ occurs during forward execution, we save the corresponding reverse address $RA_i$ in $\Phi_{RA}$ to a runtime stack in order to provide a return from the reverse of $PP_j$, $RPP_j$, to address $RA_i$ during reverse execution.

Listing 4 shows pseudo code for the stack-tracing technique. 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 reachable from multiple static locations (lines 5 and 7). The instructions inserted into the original code handle the return address bookkeeping task by saving into a runtime stack the addresses in $\Phi_{RA}$ that correspond to the addresses on the dynamically taken edges in the CG. The instructions inserted into the reverse code, on the other hand, dynamically retrieve the saved addresses from the stack and transfer the control to the retrieved addresses.

**Example 11** *Combining the reverse program partitions*: An example CG of a program is shown in Figure 19(a). Figure 19(b) shows a sample PP visit order for a specific execution instance of the program as well as the RPP visit order for the corresponding reverse execution. According to Figure 19(b), the program under consideration is forward executed starting from the beginning of $PP_1$ traversing several PPs (some twice) until ending after $PP_2$. Then, this execution is reversed by executing the corresponding reverse program starting from the beginning of $RPP_2$ (the reverse of $PP_2$) traversing several RPPs until finishing reverse execution at the end of $RPP_1$ (the reverse of $PP_1$). The RPP sequence is marked with timestamps (encircled numbers in Figures 19(b) and 19(d)) which indicate the instances when during reverse execution control is transferred between the RPPs. Note that for simplicity in this example, we assume the PPs are entered from the beginning and exited from the end (i.e., control does not leave or enter to a PP from a midpoint of the PP). Thus, the addresses on the edges of the call graph correspond to the end addresses of the PPs.

Since $PP_2$, $PP_3$ and $PP_4$ are immediately reachable from multiple static locations within the program as seen in Figure 19(a), the RCG algorithm, as seen in Figure 19(d), inserts indirect branch ("blr" − branch to link register) instructions to the end of the corresponding reverse program partitions $RPP_2$, $RPP_3$ and $RPP_4$ where the target addresses of these indirect branch instructions are determined dynamically during reverse execution. On the other hand, since $PP_5$ is immediately reachable from a single static location (from $PP_3$), an unconditional branch instruction with the hard-coded target address $RA_3$ (which is the reverse of the unique call location $A_3$ − the end of $PP_3$) is inserted to the end of the corresponding RPP, $RPP_5$ as seen in Figure 19(d).

Figure 19(d) also shows a table which indicates the dynamically determined target addresses of "blr" instructions inserted at the end of $RPP_2$, $RPP_3$ and $RPP_4$ and at what timestamp
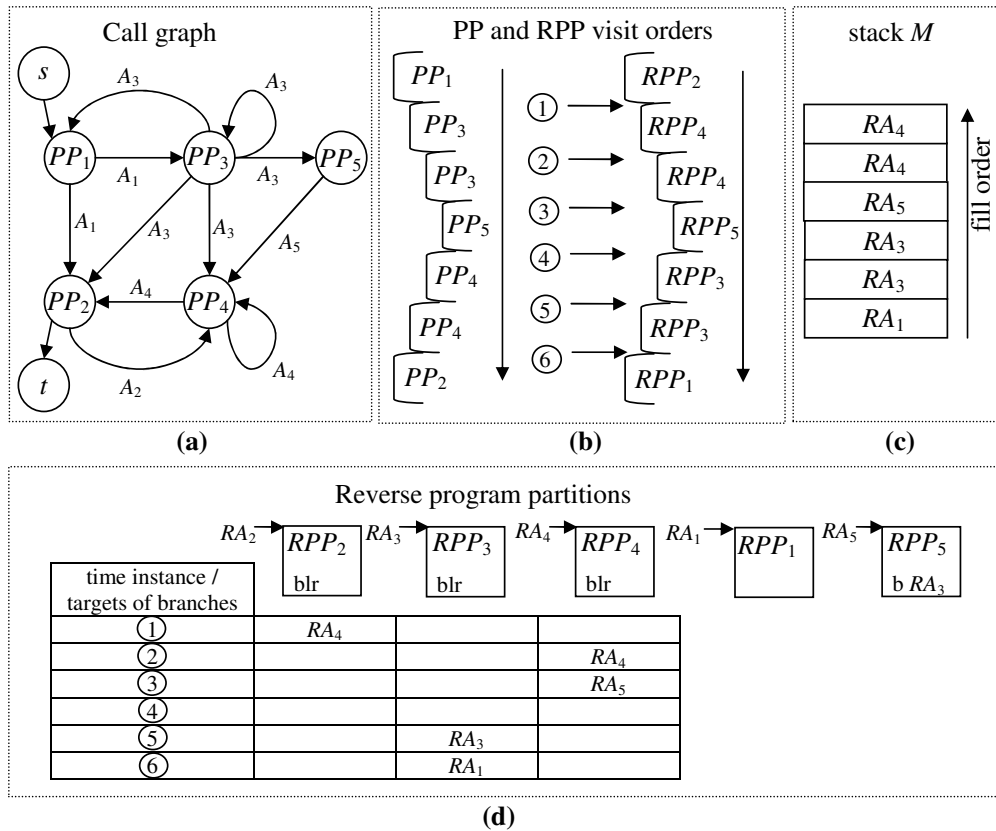
**Figure 19:** An example of combining the RPPs.

instances these addresses are determined. The second column of the table corresponds to $RPP_2$, the third column corresponds to $RPP_3$, and the fourth column corresponds to $RPP_4$.

The final state of the runtime stack $M$ at the end of forward execution of the program is shown in Figure 19(c). When a call is to be made to $PP_3$ from $PP_1$, the address $RA_1$, which corresponds to call location $A_1$ indicated by the address annotation on the edge from $PP_1$ to $PP_3$ in Figure 19(a), is entered into $M$. Then, a recursive call is made to $PP_3$ from $PP_3$ itself, and address $RA_3$ (which corresponds to call location $A_3$) is entered over the previous entry in $M$. When address $A_3$ at the end of $PP_3$ is reached again, $PP_3$ this time makes a call to $PP_5$ and the corresponding address $RA_3$ is entered into $M$ again. Similar steps are followed to enter the rest of the addresses into $M$.

60

During instruction level reverse execution, the stack-tracing technique determines the target addresses of the "blr" instructions that are placed at the end of RPPs by popping the entries from $M$ starting from the top. For instance, at the end of $RPP_2$, the top entry in $M$ of Figure 19(c) is popped. The popped address is $RA_4$; therefore, the target address of the indirect branch at the end of $RPP_2$ is dynamically set to $RA_4$ and a jump is made to $RA_4$ which is the beginning address of $RPP_4$. When the end of $RPP_4$ is reached during reverse execution, the current top entry in $M$ is popped once more. The popped address is $RA_4$; therefore, the control is directed to $RPP_4$ again. Similar steps are followed during the rest of reverse execution, which results in the correct ordering of the visits to the RPPs as seen in Figure 19(b). □

## 4.4   Summary

In this chapter, we described the implementation of the three RCG steps: program partitioning, RIG generation and RIG combination. The RCG algorithm first divides the input program into smaller regions called program partitions (PPs) by generating a partitioned control flow graph (PCFG) for each PP. Then, the RCG algorithm generates the reverses of the individual instructions (RIGs) within each PP separately. After each RIG is generated, the RIG is combined with the other RIGs generated so far. This combination process is performed hierarchically in three steps. First, the RIGs are combined into reverses of basic blocks (RBBs) by placing the RIGs in the bottom-up placement order. Next, the RBBs are combined into reverse program partitions (RPPs) by inverting the edges of the original PCFGs. Finally, the RPPs are combined into a complete reverse program by using the stack tracing technique.

In the next chapter, we describe the special implementation details of the RCG algorithm. Specifically, we describe the data structures used to generate a RIG and

explain how we treat pointer and array based memory operations. The next chapter also provides a complexity analysis of RIG generation.

# CHAPTER V

# SPECIAL RCG DETAILS

In the previous chapter, we covered the major steps of the RCG algorithm and described how a reverse program is built by analyzing and reversing the effects of individual instructions of a program. In this chapter, we explain the implementation details of reverse code generation by means of the main data structures used for redefine and extract-from-use techniques.

## 5.1 Details of RIG generation

We explained in Section 4.2 that in order to recover a value $V$, the redefine technique re-executes the instruction that originally computes $V$ provided that the inputs to the computation are still available. Similarly, the extract-from-use technique extracts $V$ from the instruction in which $V$ is used given that the other values used in the computation are still available. Therefore, the data structures to be used in these two techniques should provide us with the following information:

1. Since we seek the recovery of individual values, the program analysis for reverse code generation should be value based instead of variable based. That is, the data structures should differentiate between different values that variables attain.

2. The data structures should indicate how values flow between variables so that one can know which values are needed to recover a destroyed value.

To differentiate between values, we use the value renaming technique that is widely employed in *static single assignment* (*SSA*) form. To represent value flow between

variables, on the other hand, we use a modified version of a *value graph* [6] which we name a *modified value graph* (MVG). An MVG is different than a standard value graph because an MVG includes extra edges (pseudo definition-predicate variable edges and definition-recovering edges, explained in Sections 5.1.2 and 7.4, respectively) in addition to data dependency edges of a standard value graph. The following sections discuss the details of value renaming and MVG usage.

### 5.1.1 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) or to the combination of definitions at confluence points in a PCFG (please see value renaming for combined definitions below). Since value renaming assigns a unique name to each definition, the RCG algorithm can easily distinguish between definitions of a particular variable made at different points in a program.

#### 5.1.1.1 Value Renaming for Registers

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

**Example 12** *Value renaming for registers*: Consider the following instruction sequence:

$$addi \ \ r_2, r_1, 8 \qquad //r_2 = r_1 + 8$$

$$addi \ \ r_2, r_2, 4 \qquad //r_2 = r_2 + 4$$

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'. □

### 5.1.1.2 Value Renaming for Memory Locations

Similar to value renaming of registers, we designate different renamed values for memory locations by $m_i^j$. Here, $i$ ($i = 0, 1, 2, \ldots$) indicates the memory location being accessed, and $j$ ($j = 0, 1, 2, \ldots$) indicates the unique index of a particular renamed value residing in that memory location. Again, index $j = 0$ is always used to refer to the initial value of a memory location.

However, renaming memory values is not as easy as renaming register values. This is because the memory location being written by an instruction is not always apparent within the instruction encoding. Consequently, it might be difficult to determine whether two memory stores made by two different instructions are to the same location or not.

In case of indirect addressing, two specific values often help determine the location where an indirect store is made. These are the *base* and the *offset*. The target address of each indirect store instruction can be expressed as the summation of the base value with the offset value. If the base and offset values of a store instruction can be determined statically, then the store operation is unambiguous (e.g., a store operation for an ordinary variable, a pointer with a statically known target and an array with a statically known index).

On the other hand, if the base and/or the offset value of an indirect store instruction cannot be determined statically, then the store operation is ambiguous (e.g., a store operation for a pointer aliased to a statically unknown variable or an array with a statically unknown index).

In case of direct addressing, the target address being written by the store is apparent in the instruction encoding. Therefore, a direct store operation is unambiguous.

In the following paragraphs, we first explain how unambiguous memory stores can be distinguished, and then we explain how ambiguous memory stores are treated.
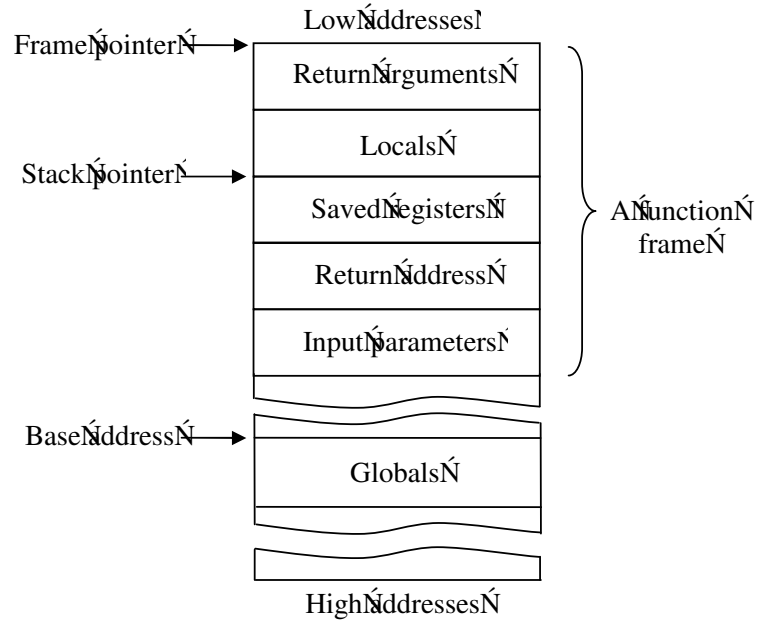


**Figure 20:** A typical memory organization made by a compiler.

Figure 20 shows a memory organization made by a typical compiler. In a typical compiler, all unambiguous indirect local stores within a PP use the frame pointer (or the stack pointer if the frame pointer is not available as a dedicated register) as the base and a statically known value (usually a constant) as the offset. All unambiguous indirect global stores in a program use the beginning address of the global data section as the base and a statically known value (again, usually a constant) as the offset [5].

Since an indirect branch instruction such as a "return from a function" both delimits a high-level program function and a PP, and since a PP is also delimited by a function call instruction that may reside inside a high-level program function, a PP is always equivalent to or a subset of a high-level program function. The important point here is that for a typical compiler, the beginning address of the global data

section is fixed throughout the execution of a program, and the value of the frame pointer is fixed during the execution of a high-level program function. Therefore, both the the frame pointer and the beginning address of the global data section are fixed throughout the execution of a PP as well. Thus, within a PP, an unambiguous indirect store can be distinguished from other unambiguous indirect memory stores by finding the fixed base address and the statically known offset being used by the store.

On the other hand, all unambiguous direct memory stores can trivially be distinguished from all other unambiguous memory stores by checking the target address in the encoding of the direct store instruction.

**Example 13** *Value renaming for unambiguous memory stores*: Consider the following instruction sequence:

$$li \quad r_1,\ 0x7000 \qquad //r_1 = 0x7000$$

$$li \quad r_2,\ 0x5000 \qquad //r_2 = 0x5000$$

$$\ldots$$

$$stw \quad r_2,\ 4(r_1) \qquad //\mathsf{mem}[r_1 + 4] = r_2$$

$$stw \quad r_5,\ 8(r_2) \qquad //\mathsf{mem}[r_2 + 8] = r_5$$

The first store instruction writes the contents of $r_2$ into the memory location at address $r_1 + 4$. The second store instruction writes the contents of $r_5$ into the memory location at address $r_2 + 8$. As seen from the first two instructions of the assembly listing above, registers $r_1$ and $r_2$ are loaded with two distinct base addresses. One of these base addresses corresponds to the stack and the other corresponds to the global data section (note that it is not essential to know which register corresponds to which base address, although the stack address is usually

67

loaded to the register being used as the frame/stack pointer – $r_1$ for the PowerPC assembly listing above). Therefore, the first store is made into address $0x7004$ and the second store is made into address $0x5008$. Since these addresses are distinct, the two memory definitions are renamed as $m_0^1$ and $m_1^1$, respectively. □

Ambiguous memory stores, on the other hand, are treated conservatively. Namely, the RCG algorithm still assigns a distinct name to an ambiguous memory definition being made by an ambiguous memory store instruction as if that definition were made into a physical memory location that had never been accessed before. However, to be conservative, the RCG algorithm also assumes that the ambiguous memory store is capable of changing the value of any other physical memory location. Therefore, all subsequent memory definitions in a PCFG that are reachable from an ambiguous memory definition are treated as destroyed. The following example illustrates how value renaming is performed for ambiguous memory stores.

**Example 14** *Value renaming for ambiguous memory stores*: Consider the following instruction sequence:

$$li \quad r_2, \ 0x5000 \qquad //r_2 = 0x5000$$

$$lwz \quad r_3, \ 4(r_2) \qquad //r_3 = \text{mem}[r_2 + 4]$$

$$lwz \quad r_4, \ 8(r_2) \qquad //r_4 = \text{mem}[r_2 + 8]$$

$$stw \quad r_6, \ 0(r_3) \qquad //\text{mem}[r_3] = r_6$$

$$stw \quad r_6, \ 0(r_4) \qquad //\text{mem}[r_4] = r_6$$

The first store instruction writes the contents of $r_6$ into the memory location pointed to by $r_3$ and the second instruction writes the contents of $r_6$ into the memory location pointed to by

$r_4$. However, neither the value of $r_3$ nor the value of $r_4$ can be determined statically as these values are loaded from some distinct memory locations by the second and the third instructions above. Therefore, the RCG algorithm renames the memory definitions made by the two store instructions as $m_0^1$ and $m_1^1$ (i.e., the definitions are assumed to be made into distinct memory locations) and the physical locations indexed as $m_0$ and $m_1$ behave as if they might coincide with any physical memory location. □

### 5.1.1.3 Value Renaming for Combined Definitions

We described in Section 4.2 that the RCG algorithm generates one or more sets where each set recovers one or more statically reaching definitions of a variable at a particular point. Moreover, we explained that these generated sets are combined via conditional branch instructions that select one of the sets according to the control flow path taken in the program. However, a typical PCFG may contain edges that join or fork at many different locations. For this reason, values defined at distinct locations in a PCFG may reach a particular point in the PCFG through multiple different paths. Therefore, in such a PCFG with a complex structure, it might be hard to track from where and under which condition(s) a definition reaches a particular point.

To ease the tracking of statically reaching definitions, the RCG algorithm, much like in the SSA form, merges different definitions reaching a confluence point under a pseudo definition and assigns a new name to the generated pseudo definition. This ensures that a unique definition of a variable reaches any point in a program. A pseudo definition generated at a confluence point selects (or dynamically represents) one of the combined definitions depending on the predicate which determines on which path the confluence point is dynamically reached. Consider the following example.

**Example 15** Figure 21 shows a portion of a PCFG. There are five different definitions of register $r_1$ reaching point **P3**: $r_1^1$ to $r_1^5$. As seen in the figure, merging of definitions at

69
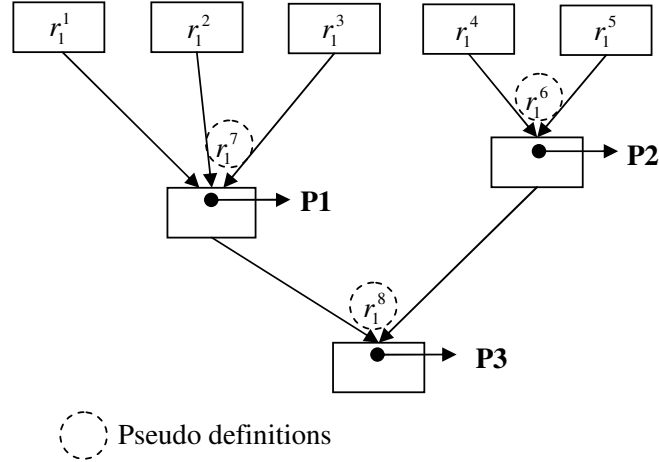
**Figure 21:** Value renaming at confluence points in a PCFG.

confluence points provides a hierarchical grouping of definitions that reach a particular point (**P1**, **P2** or **P3** in Figure 21) from several different places in the code. The definitions $r_1^1$, $r_1^2$, $r_1^3$ are merged under a new name $r_1^7$ at confluence point **P1**, and definitions $r_1^4$ and $r_1^5$ are merged under a new name $r_1^6$ at point **P2**. Finally, $r_1^6$ and $r_1^7$ are merged under the name $r_1^8$ at point **P3**. Therefore, for example, at point **P3** it is sufficient to check what definitions are combined under pseudo definition $r_1^8$ to be able to determine all the reaching definitions of $r_1$ at point **P3**. □

### 5.1.2 The Modified Value Graph (MVG)

The modified value graph provides us with the data and the control dependency information between variables. If, for instance, a register $r_1$ uses the value of register $r_2$, the value of register $r_1$ might be recoverable from $r_2$. Therefore, during the RCG analysis, the MVG answers a question such as, "Which other register still keeps the destroyed value of register $r_1$?" Another question the MVG answers is "What memory locations' and/or registers' values are needed to reconstruct the destroyed value of memory location $m_0$?" The MVG answers many additional such questions.

More formally, the modified value graph is a directed graph $G = (N, E)$ where $N$ is the set of nodes and $E$ is the set of edges between those nodes. Each node in an MVG corresponds to a definition of a register or a memory location (renamed by a value renaming operation) including the pseudo definitions as mentioned in the previous section. In an MVG, we designate with $\mu$ the pseudo definitions that are generated at loop entrances (at points in a PCFG where at least one forward edge and at least one backward edge join) and with $\phi$ the pseudo definitions that are generated at any other confluence point.

**Example 16** *Nodes in an MVG*: Figure 22(a) shows an example PP. The PCFG and the MVG for this PP are shown in Figures 22(b) and 22(c), respectively. Figure 22(c) indicates (via arrows) at which point in the PP each definition is made. For example, instruction "$lwz\ r_2,\ 0(r_1)$" writes a value into register $r_2$ and thus defines $r_2^1$. Note that the definitions $m_0^0$, $r_1^0$, $r_4^0$ and $r_5^0$ are the initial definitions of the corresponding locations and are defined prior to the PP shown in Figure 22 (the location $m_0$ is the memory location at the address value contained in $r_1$). Therefore, the nodes for these initial definitions in the MVG shown in Figure 22(b) do not correspond to any instruction in the PP shown in Figure 22(a).

The node indicated with $\phi$ in Figure 22(b) corresponds to the pseudo definition $r_3^3$ generated at the confluence point of edges of the PCFG shown in Figure 22(c). Similar to SSA form, we say $r_3^3 = \phi(r_3^1, r_3^2)$ meaning that $r_3^3$ is a pseudo $\phi$-definition which selects either $r_3^1$ or $r_3^2$ (depending on the predicate $r_2^1 \geq 0$). Note that since each definition is represented by a *single* node in an MVG, the size of an MVG is bounded by the number of definitions in a program. $\square$

The edges of an MVG, on the other hand, represent the data and the control dependency relationships between variables in a program. Therefore, the edges indicate on which other registers and/or memory locations a register or a memory location is
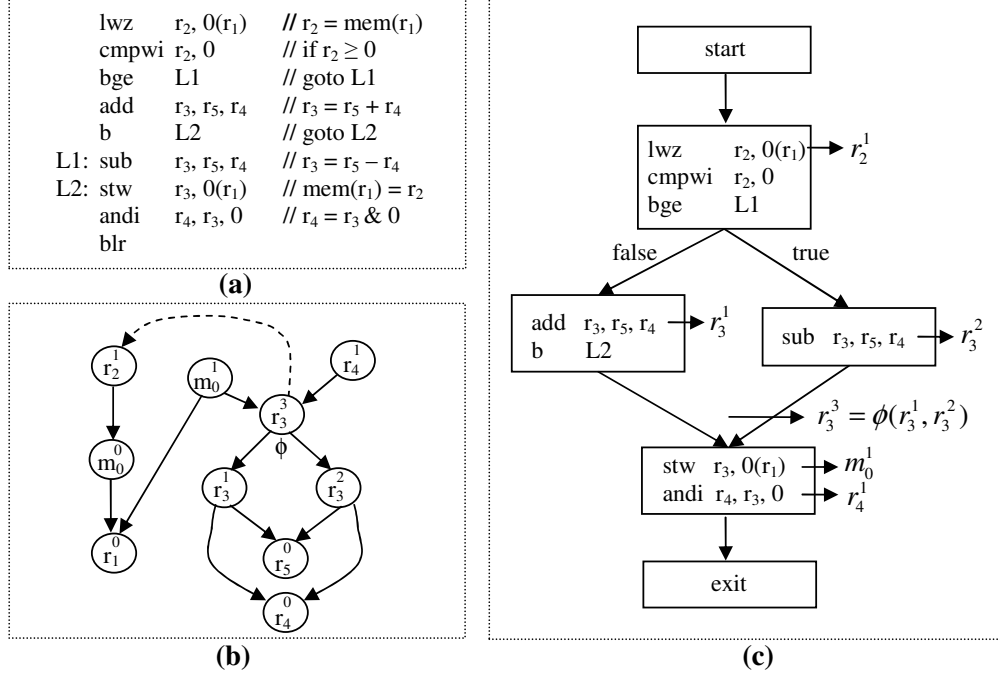
**Figure 22:** (a) An example PP. (b) Corresponding MVG. (c) Corresponding PCFG.

data or control dependent. There are three kinds of edges in an MVG. While the first two kinds of edges are mainly used during reverse code generation, the third kind of edge is used in program slicing. Therefore, we will explain only the first two kinds of edges for now. The third kind will be explained in Chapter 7 when dynamic slicing support is described and will be called a *definition-recovering* edge.

1. *Use-definition edge*: This type of edge extends from a node $n_i$ to another $n_j$ that is used by $n_i$. A node $n_i$ can use another node $n_j$ in three different ways: (1) $n_i$ and $n_j$ are the values for target and source operands of an instruction $\alpha$, respectively (i.e., the computation of $n_i$ uses the value in $n_j$); (2) $n_i$ is a memory value and $n_j$ is a register value determining the memory address of $n_i$; or (3) $n_i$ is a pseudo definition node and $n_j$ is one of the definitions that is being combined under that pseudo definition.

2. *Pseudo definition-predicate variable edge*: A pseudo definition-predicate variable edge extends from a $\mu$ node or a $\phi$ node to the nodes of definition(s) used in the controlling predicate expression for the $\mu$ node or the $\phi$ node (as explained above, the $\mu$ nodes and the $\phi$ nodes select one of their combined definitions according to a predicate). Therefore, this type of edge indicates which other register or memory location controls which definition a $\mu$ node or a $\phi$ node selects. Thus, a pseudo definition-predicate variable edge is used for extracting the control dependency information between the nodes.

**Example 17** *Edges in an MVG*: Figure 22(b) shows use-definition edges as solid lines. For instance, the edge from $m_0^1$ to $r_3^3$ in Figure 22(b) indicates that $m_0^1$ uses the definition $r_3^3$ (actually, $r_3^3$ is stored into $m_0^1$). This edge is an example of case (1) in the above definition of a use-definition edge. Similarly, the edge from $m_0^1$ to $r_1^0$ indicates the memory address of $m_0^1$ is determined by $r_1^0$. This edge is an example of case (2) in the definition of a use-definition edge. Finally, the edges from *phi* node $r_3^3$ (i.e., $r_3^3 = \phi(r_3^1, r_3^2)$) to the combined definitions under $r_3^3$, namely the definitions $r_3^1$ and $r_3^2$, are examples of case (3) in the definition of use-definition edge.

On the other hand, a pseudo definition-predicate variable edge is shown as the dotted line in Figure 22(b). This pseudo definition-predicate variable edge connects the pseudo definition $r_3^3$ to the definition $r_2^1$. This is because which combined definition $r_3^3$ selects (i.e., $r_3^1$ or $r_3^2$) is controlled by the predicate expression $r_2^1 \geq 0$ which contains $r_2^1$ as the only predicate variable. □

An MVG is constructed in the same way as a standard value graph described in [39]. Therefore, we will not go into details of MVG construction in this thesis.

Having described the nodes and the edges in an MVG, we next explain how an MVG is used in the redefine and the extract-from-use techniques.

### 5.1.3   Recovery of a Destroyed Definition Using an MVG

Each register or memory definition made in a program is represented by a node in an MVG, and the dependency information between the definitions in a program is represented by the MVG edges. Therefore, using this dependency information, the RCG algorithm traverses an MVG and tries to locate the nodes that might be used to reconstruct or extract a destroyed value by the redefine and the extract-from-use techniques, respectively.

A node $n_i$ in an MVG can have at most one of the following attributes at a point $P$ in a program: *killed, available* or *partially-available.* Node $n_i$ is killed at $P$ if the definition $n_i$ represents does not reach $P$; $n_i$ is available at $P$ if the definition $n_i$ represents reaches $P$ along all paths; and $n_i$ is partially available at $P$ if the definition $n_i$ represents reaches $P$ along a proper subset of all paths (e.g., along a path controlled by a predicate expression), i.e, $n_i$ represents one of the definitions combined under a pseudo definition.

As seen in Figure 23, suppose that an instruction $\alpha$ destroys the value of a variable $V$ at a particular point in a PP. Let us name the point just before and after $\alpha$ as $\mathbf{P}$ and $\mathbf{P'}$, respectively. As explained in Section 4.2, in order to recover the value of $V$, *Gen_RIG()* finds the reaching definition(s) $\delta$ of $V$ at point $\mathbf{P}$ by calling *Find_Reaching_Def()* at line 3 of Listing 2 (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 as shown by definition $\delta$ in Figure 23). Then, *Gen_RIG()* finds in the MVG the node that corresponds to the found pseudo or reaching definition. Suppose that the found node is $n_i$ (i.e., $n_i$ is the node that represents the reaching definition(s) $\delta$ of $V$ at point $\mathbf{P}$). Since $\delta$ is destroyed by $\alpha$, node $n_i$ is killed at point $\mathbf{P'}$. Now, if one or both of the following are true at $\mathbf{P'}$, *Gen_RIG()* can recover $n_i$ (i.e., the definition(s) $\delta$) by generating the appropriate
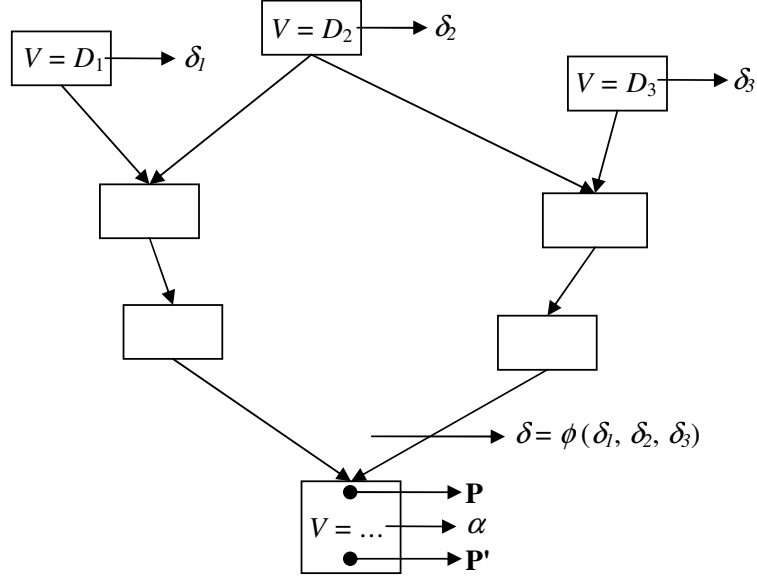
**Figure 23:** An instruction modifying a variable $V$ at a point in a PP.

instructions (note that in the rest of this section, we interchangeably use the term "node" to refer to a node $n$ in an MVG and to the definition $n$ represents).

(a) Each $n_j$, where there exists a use-definition edge from $n_i$ to $n_j$ (i.e., $n_j$ is a child of $n_i$), is available, and an instruction $\beta$ defines $n_i$ (see Figure 24(a)). Please note that $\beta$ needs each $n_j$ in order to be potentially re-executed.

(b) An $n_k$, for which there exists a use-definition edge from $n_k$ to $n_i$ (i.e. $n_k$ is a parent of $n_i$), is available, and each $n_s$, $n_s \neq n_i$, for which there exists a
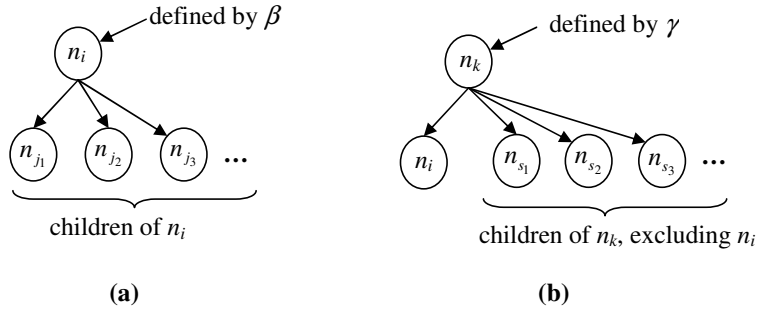


**Figure 24:** (a) Recovering a node from its children. (b) Recovering a node from one of its parents and corresponding siblings.

use-definition edge from $n_k$ to $n_s$ (i.e., $n_s$ is a sibling of $n_i$ for parent $n_k$), is available as well. Moreover, an instruction $\gamma$ defines $n_k$ by using $n_i$ as a source operand, and the opcode of $\gamma$ allows extraction of $n_i$ out of $\gamma$ (see Figure 24(b)). Please note that $\gamma$ needs each $n_s$ in order to potentially extract $n_i$.

If (a) holds, $n_i$ can be recovered at $\mathbf{P'}$ by executing $\beta$ without any change (i.e., by the redefine technique). On the other hand, if (b) holds, $n_i$ can be recovered at $\mathbf{P'}$ by extracting $n_i$ out of $\gamma$ (i.e., by the extract-from-use technique).

There are also two special conditions that enable the recovery of $n_i$ even if neither case (a) nor case (b) holds.

(c) In case (b) above, at least one node $n_m$ among $n_k$ and the nodes $n_s$ that are required to recover $n_i$ is partially-available controlled by a predicate expression $\Upsilon_m$, and the rest of the nodes among $n_k$ and the nodes $n_s$ are available. Moreover, the variables in $\Upsilon_m$ are also available.

(d) $n_i$ is a node that represents a pseudo definition (e.g., $\delta$ in Figure 23) where (a) holds for each definition combined under $n_i$ (e.g., $\delta_1$, $\delta_2$ and $\delta_3$ in Figure 23) and each node $n_p$ where there exists a pseudo definition-predicate variable edge from $n_i$ to $n_p$ is available.

If (c) holds, then $n_i$ is recoverable by extracting $n_i$ from $\gamma$ only if $\Upsilon_m$ evaluates to true (i.e., only if the program reaches $\mathbf{P'}$ via the control flow path(s) along which $\Upsilon_m$ is true). Since the recovery of $n_i$ depends on a predicate expression to become true, we state that $n_i$ is partially-recoverable by $\gamma$ at point $\mathbf{P'}$. To recover $n_i$ fully, $n_i$ must be partially-recoverable by other such instructions $\gamma_q$ for all other possible paths to $\mathbf{P'}$. If this is the case, $n_i$ is recovered by combining the instructions $\gamma_q$ each of which partially recovers $n_i$ on a particular set of paths to $\mathbf{P'}$.

On the other hand, if (d) holds, then $n_i$ can be recovered by recovering each definition combined under $n_i$ (e.g., $\delta_1$, $\delta_2$ and $\delta_3$ in Figure 23).
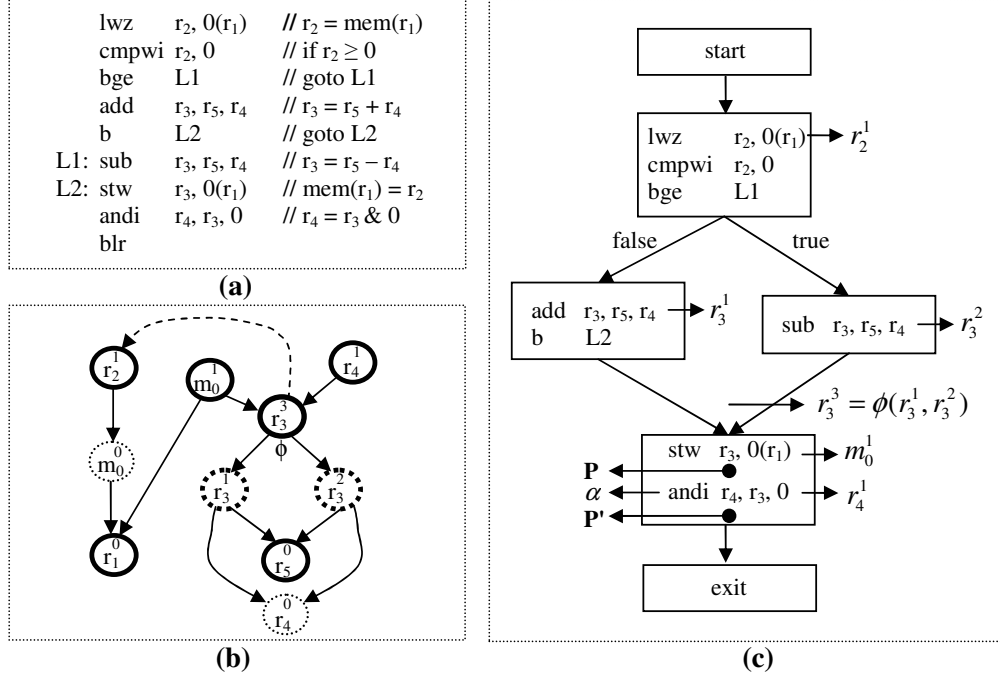
**Figure 25:** (a) An example PP. (b) Corresponding MVG. (c) Corresponding PCFG.

Finally, the above actions can be applied recursively, that is, if a node that is required to recover $n_i$ is killed, then $n_i$ might still be recovered by recovering the killed node 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 a state saving instruction to recover the killed node.

**Example 18** Figure 25 shows the example PP, the corresponding MVG and the corresponding PCFG previously shown in Figure 22. In Figure 25(b), available nodes at point $\mathbf{P'}$ are encircled with solid thick lines, partially-available nodes at point $\mathbf{P'}$ are encircled with dotted thick lines, and killed nodes at point $\mathbf{P'}$ are encircled with dotted thin lines. Suppose that we would like to generate a RIG for the instruction $\alpha = addi\ r_4,\ r_3,\ 0$. This instruction $\alpha$ modifies register $r_4$; thus, the RIG for $\alpha$ should recover $r_4$. The reaching definition of register $r_4$ at point $\mathbf{P}$ is $r_4^0$. Since $r_4^0$ does not have any children, case (a) explained above does not hold. Since neither of the parents of $r_4^0$ are available, case (b) does not hold, either. However, case (c) holds for

both parents of $r_4^0$. Therefore, the definition $r_4^0$ can be partially recovered by using the parent node $r_3^1$ and and the available sibling node $r_5^0$ for the predicate expression $r_2^1 \geq 0$. The same node ($r_4^0$) can also be partially recovered by using the other partially-available parent node $r_3^2$ and the available sibling node $r_5^0$ for the complementary predicate expression $r_2^1 < 0$ (note that we assume the computation of neither $r_3^1$ nor $r_3^0$ results in an overflow/underflow). Since $r_4^0$ is partially recoverable for all possible control flow paths reaching $\mathbf{P'}$, $r_4^0$ is fully recoverable. In this case, the RIG for $\alpha$ will be as follows:

```
        cmpwi r₂, 0

    bge     L1

    sub     r₄, r₃, r₅

    b       L2

L1:  sub    r₄, r₅, r₃

L2:  ...
```

$\square$

Listing 5 shows the code snippet for recovering a definition $\delta$ using an MVG. The *Recover()* function shown in Listing 5 starts from the MVG node $n_i$ of $\delta$ that is to be recovered and traverses the MVG by recursively calling itself at various places. Note that each recursion of *Recover()* is provided with a set $\zeta_\delta$ which includes the instructions to recover $\delta$ on a particular set of paths reaching to the point where $\delta$ is destroyed (see Section 4.2 for a detailed description of such a set $\zeta_\delta$). At the end of each recursion of *Recover()*, $\zeta_\delta$ is appended with a new set of reverse instructions $\zeta_{sel}$ which recovers $\delta$ on a new set of paths. As explained in Section 4.2, *Recover()* is called recursively as many times as necessary to cover all paths to the point where $\delta$ is destroyed.

---

**Listing 5** *Recover()*: Recovering a definition

---

**Input**: An MVG, a definition $\delta$ to be recovered in the MVG and a set of reverse instruction $\zeta_\delta$ from the previous recursion of *Recover()*

**Output**: $\zeta_\delta$ appended with reverse instructions from the current recursion

**begin**

1   $n_i = \text{node\_of}(\delta)$

2   **if** $n_i ==$ available **then**

3     $RIG_\delta.\text{Cost} = 0$

4     return

5   **end if**

6   $Candidate\_parent\_sets = \phi$

7   $Candidate\_child\_sets = \phi$

8   **for all** $n_k \in Parents(n_i)$ **do**

9     $Recover(definition\_of(n_k),\ \zeta_p^k)$

10    **if** $\zeta_p^k.\text{Cost} \mathrel{!=} \infty$ **then**

11      **for all** $n_s \in Siblings(n_i, n_k)$ **do**

12       $Recover(definition\_of(n_s),\ \zeta_p^k)$

13        **if** $\zeta_p^k.\text{Cost} == \infty$ **then**

14         break

15        **end if**

16      **end for**

17      **if** $\zeta_p^k.\text{Cost} \mathrel{!=} \infty$ **then**

18       $Candidate\_parent\_sets \mathrel{+}= \zeta_p^k$

19      **end if**

20    **end if**

21   **end for**

22   **for all** $n_j \in Children(n_i)$ **do**

23     $Recover(definition\_of(n_j),\ \zeta_c^j)$

24    **if** $\zeta_c^j.\text{Cost} = \infty$ **then**

25     break

26    **end if**

27    **if** $\zeta_c^j.\text{Cost} \mathrel{!=} \infty$ **then**

28      $Candidate\_child\_sets \mathrel{+}= \zeta_c^j$

29    **end if**

30   **end for**

31   $\zeta_p^f = Extract\_from\_use(\text{Candidate\_parent\_sets},\ n_i)$

32   $\zeta_c^f = Redefine(\text{Candidate\_child\_sets},\ n_i)$

33   $\zeta_{sel} = Min\_size(\zeta_c^f,\ \zeta_p^f)$

34   $\zeta_\delta \mathrel{+}= \zeta_{sel}$

35   $\zeta_\delta.\text{Cost} \mathrel{+}= \zeta_{sel}.\text{Cost}$

**end**

---

First, to be able to recover $n_i$ from a parent node $n_k$ using the extract-from-use technique, *Recover()* recovers $n_k$ (line 9) and all sibling nodes of $n_s$ used by parent $n_k$ (line 12). These sibling nodes correspond to the definitions $n_k$ uses other than $n_i$ and are needed to extract $n_i$ from $n_k$. This process is repeated for each parent node of $n_k$ (lines 7 to 20). Second, *Recover()* recovers each child node $n_j$ of $n_i$ to redefine $n_i$ using its children (lines 22 to 30).

Next, the sets of instructions (denoted by *Candidate_parent_sets*) obtained by recovering each parent $n_k$ and the corresponding siblings $n_s$ of $n_i$ are combined into a final set, $\zeta_p^f$ using predicate expressions if needed (line 31). Similarly, the sets of instructions (denoted by *Candidate_child_sets*) obtained by recovering each child $n_j$ of $n_i$ are combined into another final set, $\zeta_c^f$ again by using predicate expressions if needed (line 32). Finally, *Recover()* selects among $\zeta_c^f$ and $\zeta_p^f$ the minimum sized set to be appended to the return set, $\zeta_\delta$ (line 34), and adds to the cost of $\zeta_\delta$ the cost of the selected set (line 35).

## 5.2 Complexity of RIG Generation

This section makes a complexity analysis of RIG generation using an MVG. Let us designate with $N$ the number of nodes in the MVGs generated for all the PPs in a program and with $M$ the average degree of a node in an MVG (i.e., how many neighbors a node has). Also, let us designate with $K$ the maximum number of repetitive applications of the redefine and the extract-from-use techniques allowed. As shown in Listing 5, to recover a node $n_i$, the RCG algorithm traverses the neighbors of $n_i$ searching for available or partially-available nodes (defined in Section 5.1.3) until all dependencies of $n_i$ are resolved. Moreover, this search is recursively repeated for all other killed neighbors of $n_i$. Therefore, in the worst case, if we assume that all nodes to which there exists a path from a node $n_i$ in an MVG are killed, the RCG algorithm has to visit at most $M^K$ nodes to recover $n_i$. Assuming that this process is repeated

to recover each node in an MVG, the complexity of RCG turns out to be proportional with $N \times M^K$.

Since an MVG is constructed from the nodes within a PP only, $M$ is dependent only on the size of a PP but not on the size of a complete program. Therefore, in a typical program with a fixed PP size, $M$ is also fixed. Therefore, the complexity typically grows linearly with $N$. (Note that $N$ typically grows linearly with code size.)

**Example 19** Assume that a program $T$ is composed of one million instructions; therefore, $N$, the number of nodes in $T$, will be approximately one million. Also, assume that $M$ is 10. This means that a node has 10 neighbors in average. If $K$ is taken to be 3, the number of iterations to recover each node will be $1,000,000 \times 10^3 = 10^9$. Assuming that a processor can execute $10^9$ iterations per second, the total execution time will be 1 second. □

## 5.3 Summary

In this Chapter, we explained the implementation details of the RCG algorithm by explaining the main data structures used for generating a RIG. We also provided a complexity analysis of RIG generation using an MVG. In the next chapter, we provide a summary of the overall RCG algorithm.

# CHAPTER VI

# SUMMARY OF THE OVERALL RCG

# ALGORITHM

We presented the three main steps of the RCG algorithm and explained the details behind our reverse code generation technique. Before going into the explanation of the extensions we made to enrich the RCG algorithm with dynamic slicing support, it would be better to briefly go over the main points that have been explained so far. In this chapter, we give a summary of the overall RCG algorithm by the help of the flowchart shown in Figure 26.

Given a program at the assembly instruction level, the RCG algorithm first divides the program into program partitions (by building the PCFGs) and constructs a CG of the program (Box 1 in Figure 26 – note that the algorithmic details implementing Box 1 have been described previously in Listing 1 shown in Section 4.1 and in Listing 3 shown in Section 4.3.3.1). Then, the RCG algorithm enters a main loop where the instructions of each program partition are read one after another and the reverse program partitions are built.

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 (Box 3). The algorithm for Box 3 has been given in Listing 2 in Section 4.2.

After the RCG algorithm generates a RIG for an analyzed instruction in a BB, the RCG algorithm places that RIG into the corresponding RBB in bottom-up placement order as described in Section 4.3.1 (Box 4).
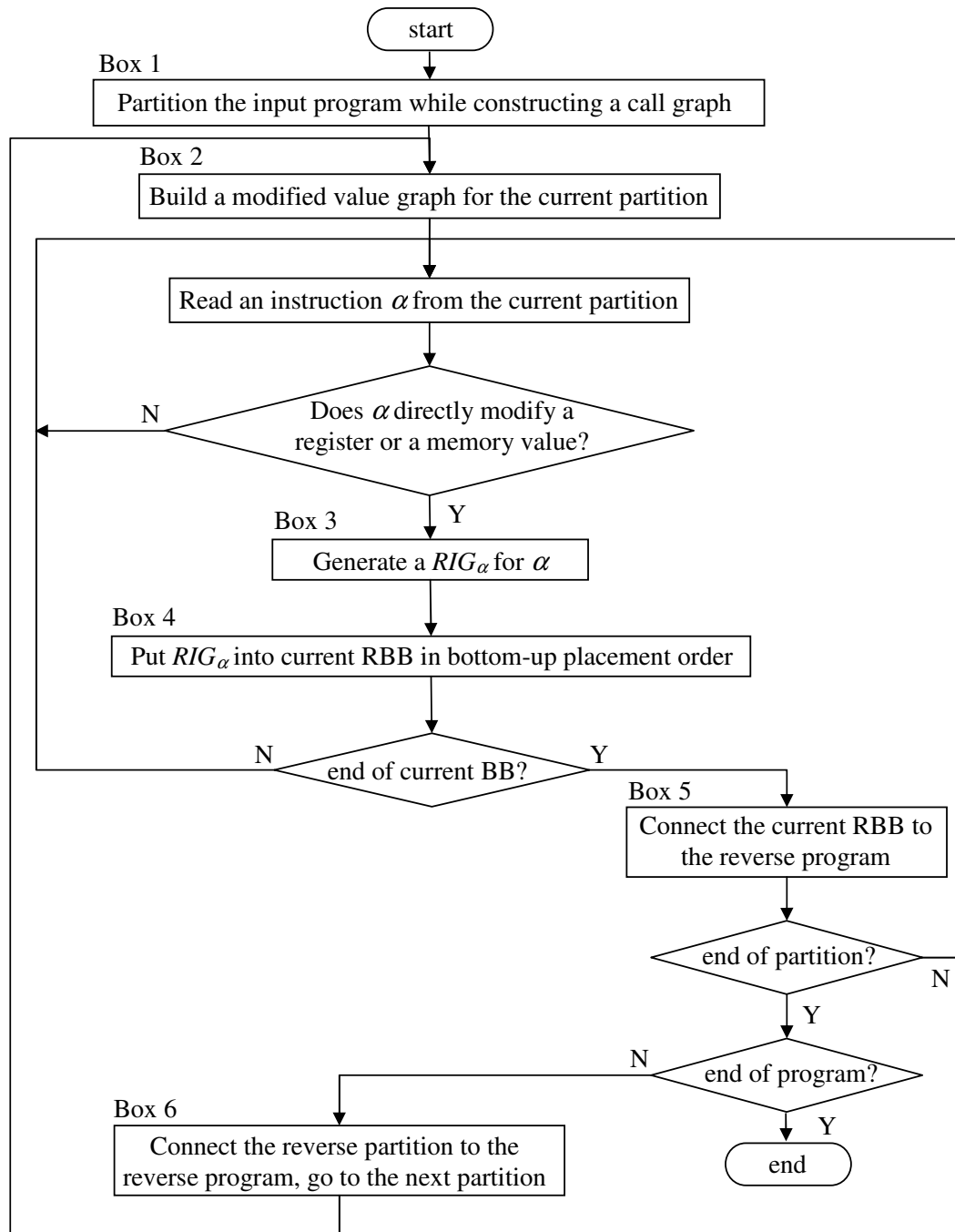
**Figure 26:** A high-level flowchart of the RCG algorithm.

When the RCG algorithm reaches the end of a basic block, the RCG algorithm connects the reverse of that basic block to rest of the reverse program as explained in Section 4.3.2 (Box 5). Similarly, when the RCG algorithm completes the construction of the current reverse program partition, the RCG algorithm connects the constructed reverse program partition to the rest of the reverse program (Box 6) and moves on to the construction of the next reverse program partition (the implementation of Box 6 has been described in Listing 4 in Section 4.3.3). This process is repeated until the end of the program is reached.

**Example 20** *Reverse Code Generation*: Consider the Huffman encoder application shown in Figure 27(a). This application is composed of six high-level functions: *main*, *build_huffman_tree*, *find_huffman_codes*, *build_huffman_table* and *write_huffman_header*, *write_output_data*. When the RCG algorithm executes Box 1 in Figure 26, the Huffman encoder application is divided into ten PPs as shown in Figure 27(a) and a call graph shown in Figure 27(b) is constructed. Then, the RCG algorithm enters a loop where it builds an MVG for each PP of Huffman encoder by executing Box 2 in Figure 26 and generates the RPP for that PP. The number in parentheses next to $PP_i$ indicates the number of nodes in the MVG generated for $PP_i$. The reverse of a PP is generated by reading the instructions in the PP one by one and by constructing (Box 3) and combining (Box 4 and Box 5) the corresponding RIGs. Within the RPP generation loop, Box 3 and Box 4 are executed 314 times, while Box 5 is executed 44 times. Finally, the RCG algorithm combines the RPPs generated for each PP to generate the reverse of Huffman encoder. □
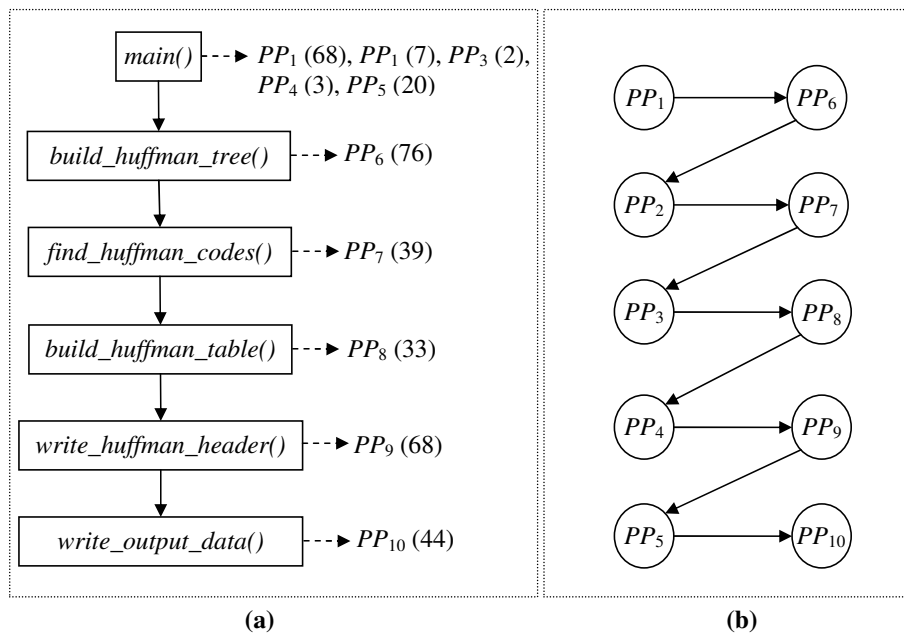
**Figure 27:** (a) Huffman encoder block diagram showing PPs for each function and MVG node count for each PP (b) Corresponding call graph.

# CHAPTER VII

# DYNAMIC SLICING SUPPORT

## *7.1   Background*

In order to understand the benefits of dynamic slicing for assembly level reverse execution, we need to first understand what program slicing performs. This section gives an overview of program slicing.

**Definition 7.1.1** *Static slice*: A static slice of a program is a set of program statements which *may* influence the value of a variable $V$ at a statement $S$. Variable $V$ and statement $S$ comprise the slicing criterion which we designate as $C = (V,S)$. □

**Definition 7.1.2** *Dynamic slice*: A dynamic slice of a program is a set of program statements which affect the value of a variable $V$ at a specific execution instance $q$ of a statement $S$ given a set of program inputs $X$. We designate a dynamic slicing criterion as $C = (X, V, S^q)$. □

Since we are interested in assembly level reverse execution, a dynamic slice of a program should be at the assembly level as well. Thus, for the first time ever (to the best of this author's knowledge), we can define a dynamic slice to apply to an assembly level program as follows.

**Definition 7.1.3** *Assembly level dynamic slice*: An assembly level dynamic slice of a program is a set of assembly instructions which affect the value of a register or a memory location

$(L)$ at a specific execution instance $q$ of an instruction $I$ given a set of program inputs $X$. We show an assembly level dynamic slicing criterion as $C = (X, L, I^q)$. □

An instruction influences the value of a register or a memory location another instruction modifies if there is a direct or an indirect (i.e., transitive) dependency between those two instructions. A direct dependency between two instructions can be either a data dependency or a control dependency. If an instruction $I_k$ uses a register or a memory location $L$ that is defined by another instruction $I_k$ and $L$ is not subsequently overwritten before being used by $I_k$, then $I_k$ is data dependent on $I_j$. If the execution of $I_k$ depends on the boolean outcome of $I_j$, then $I_k$ is control dependent on $I_j$. On the other hand, an indirect dependency between two instructions $I_k$ and $I_j$ happens if $I_k$ is directly dependent on another instruction $I_m$ and $I_m$ is directly dependent on $I_j$. Therefore, the transitive closure of dependencies of an instruction $I_k$ gives all the instructions that may influence the value of $L$ and thus constitutes a static slice with respect to $L$ [29, 41, 47].

Although any static slice of a program can be extracted by a pure static analysis, the extraction of a dynamic slice requires runtime information of a program. This information captures what control flow path the program follows to reach the specific instance of the instruction in the slicing criterion. It may very well be the case that some instructions, although part of the static slice with respect to a static slicing criterion $C = (L, I)$, cannot influence the value in $L$ due to a lack of a dynamically taken path between those instructions and $I$. Therefore, dynamic slicing removes such instructions from a static slice producing a more compact and precise slice.

## 7.2 Overview of Slicing Approach

Assembly level reverse execution along a dynamic slice can be defined as a partial reverse execution method which visits only the instructions that are in the dynamic slice

and which recovers only those values that are relevant to the dynamic slice instructions (i.e., the values that are used or generated by the dynamic slice instructions). In this section, we give an overview of our methodology to achieve such a partial and potentially faster reverse execution.

While trying to implement assembly level reverse execution along a dynamic slice, one could ask the following question: "If the RCG algorithm gives us the reverse of any code that is input to it and if we want to reverse execute along a dynamic slice only, why not just extract the desired dynamic slice from a program first and then use the RCG algorithm to generate the reverse of that slice?" Although, at first sight, this approach seems to provide a trivial solution to our problem, it does not serve our purpose. This is because some values that are relevant to the slice instructions can only be recovered by undoing the instructions that are out of the slice. Therefore, reversing the instructions within a slice only is not sufficient to provide reverse execution along that slice. Consider the following example.

**Example 21** Figure 28 shows a PowerPC assembly code piece with five instructions. The instructions in the dynamic slice that is calculated according to the slicing criterion $C = (r_1 = 0, r_3, (\text{addi } r_3, r_3, 1)^1)$ are enclosed in rectangles. In other words, the enclosed instructions are the instructions that influence the value of $r_3$ at the first instance of instruction "addi $r_3$, $r_3$, 1" when $r_1$ initially contains value '0'. Suppose that the program counter is currently at position **P5** and we would like to reverse execute the program back to point **P1** by following the dynamic slice under consideration. This implies we first need to jump to **P2** and then jump to **P1** bypassing points **P3** and **P4**. These points are bypassed because the third and the fourth instructions are not within the dynamic slice. While reverse executing the program through this path, we expect to retrieve the values relevant to the instructions on the path. For instance, when we reach point **P1**, we should have retrieved the values of $r_2$ and $r_1$

88

P1 →  addi    $r_2, r_1, 2$   // $r_2 = r_1 + 2$

P2 →  mulli   $r_3, r_2, 4$   // $r_3 = r_2 \times 4$

P3 →  addi    $r_4, r_1, 1$   // $r_4 = r_1 + 1$

P4 →  divw    $r_2, r_{1,} r_4$   // $r_2 = r_1 / r_4$

P5 →  addi    $r_3, r_3, 1$   // $r_3 = r_3 + 1$

**Figure 28:** A code piece and a dynamic slice.

because both of these values are relevant to the first instruction in the dynamic slice. On the other hand, at point **P1** we do not care about the value of $r_4$ because this value is neither used nor generated by the instructions in the dynamic slice.

However, if we were to reverse execute only the instructions within the dynamic slice, we would not have retrieved the value of $r_2$ at point **P1** because $r_2$ is overwritten by the fourth instruction which is outside the slice. Therefore, while trying to obtain the values relevant to the instructions within the slice, we might have to undo instructions that are outside the slice.

□

As Example 21 illustrates, we should extend the RCG algorithm to take into account all the instructions in a program in order to determine which instructions to undo and which instructions to skip. In doing so, we choose to remove from a complete reverse program the instructions that are unnecessary for reverse execution along a particular dynamic slice. Hereafter, we refer to the RCG algorithm with dynamic slicing support as *RCG with Slicing* or *RCGS*.

Figure 29 shows a high level view of our methodology. Given an input program compiled to assembly, we first generate the corresponding complete reverse program using the RCG algorithm we presented in Chapter 4. Then, the programmer can start debugging the program with full-scale reverse execution support. When the

89

programmer decides to obtain a dynamic slice to investigate the roots of a bug in the program, the RCGS algorithm performs its extended static analysis. This static analysis, when combined with the runtime debugger support, enables the user to reverse execute the program along the desired dynamic slice. Therefore, reverse execution along a dynamic slice can be achieved without having to restart the program under consideration.

The next section presents the extensions we made to the RCG algorithm following the instruction removal approach explained above.

## 7.3   The Extensions to the RCG Algorithm

As explained in the previous section and also shown in Figure 29, the extensions to the RCG algorithm for dynamic slicing support are composed of two parts. The first part is the static analysis part. The second part is debugger support which incorporates dynamic information. We describe each part in the following subsections.



Gray arrows indicate the base static analysis performed only once per program
White arrows indicate the extended static analysis performed for each dynamic slice
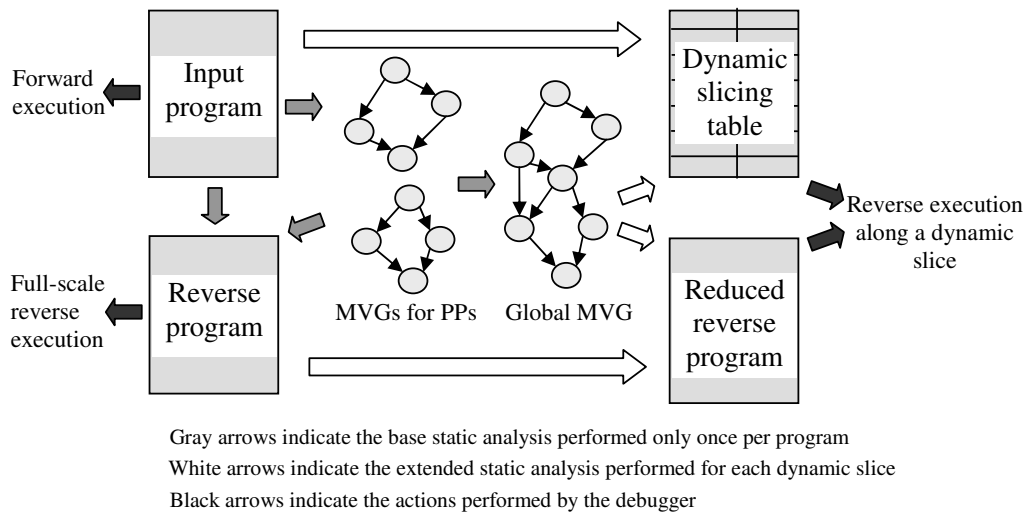Black arrows indicate the actions performed by the debugger

**Figure 29:** A diagram of the RCGS algorithm.

### 7.3.1 The Static Analysis Part

The extended static analysis is performed to generate a *reduced reverse program* and a *dynamic slicing table* both of which provide reverse execution along a particular dynamic slice (Figure 29). The reduced reverse program is a reverse program which excludes the instructions that are *definitely* known to be unnecessary for reverse execution along the dynamic slice under consideration. When executed, the reduced reverse program recovers the program state relevant to the corresponding dynamic slice.

The RCGS algorithm performs the extended static analysis over a *global MVG* (see Section 5.1.2) which is obtained by combining the MVGs of individual PPs. This combination process is performed by adding data dependency edges between the nodes of individual MVGs using a standard global data-flow analysis [39].

A global MVG for a program is generated only once after the generation of a reverse program. Afterwards, the generated global MVG can be used for the generation of different reduced reverse programs each of which enables reverse execution along a different dynamic slice. Given a dynamic slicing criterion $C = (X, L, I^q)$, we first use the global MVG to statically find the definitions which might influence $L$ at $I$. We call such definitions the *potentially influencing definitions*.

A reduced reverse program should be composed of only those instructions that are required to recover the potentially influencing definitions. Ideally, these instructions are the RIGs that recover the potentially influencing definitions. However, as explained in Section 4.2, to recover a definition, we might have to use some other definitions as well. Therefore, the reduced reverse program also includes those RIGs that recover these extra definitions. We collectively call the potentially influencing definitions and their recovering definitions the *essential definitions*.

In the next section, we introduce the debugger support which is the second part of the RCG extensions. The details of how a global MVG is used to determine the

potentially influencing definitions and to extract a reduced reverse program will be explained in Section 7.4.

### 7.3.2  Debugger Support

Debugger support reveals the control flow along the chosen dynamic slice in the input program. The debugger uses the dynamic slicing table to map the position in the reduced reverse program to the position in the input program at runtime. This is achieved by using the correspondence between the input program and the reduced reverse program. Since the reduced reverse program undoes only those instructions that are actually executed during forward execution (the remaining instructions are bypassed by the control flow predicates in the reverse code), the runtime information required for building a dynamic slice is reconstructed during reverse execution rather than being collected during forward execution.

## 7.4  Generating a Reduced Reverse Program Using a Global MVG

The information required to find the essential definitions in a program is obtained from the edges of a global MVG. In this process, we also utilize a third type of edge (in addition to use-definition and pseudo definition-predicate variable edges explained in Section 5.1.2) whose definition was not included in Section 5.1.2.

3. *Definition-recovering edge*: As the name indicates, a definition-recovering edge combines a killed definition $\delta_i$ with the definition $\delta_j$ that is used to recover $\delta_i$. The solid thick lines in Figure 30(b) are examples of this kind of edge. For instance, as explained in Example 18, the node $r_4^0$ is recovered using nodes $r_3^1$, $r_3^2$ and $r_5^0$. Therefore, three definition-recovering edges are placed from node $r_4^0$ to node $r_3^1$, node $r_3^1$ and node $r_5^0$, respectively. This type of edge helps
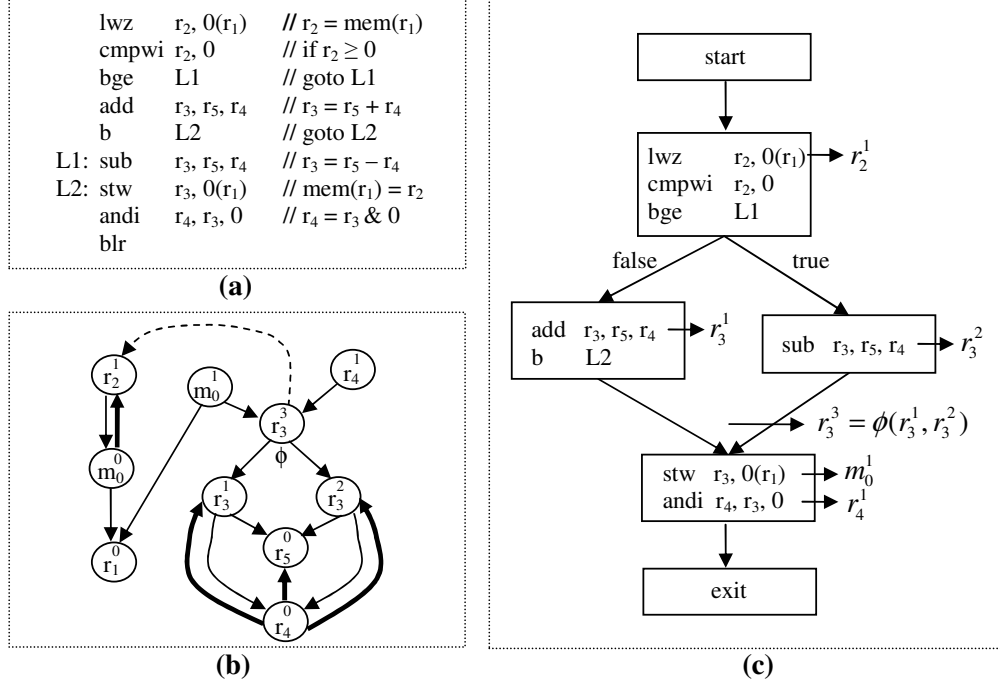
**Figure 30:** (a) An example program. (b) Corresponding global MVG.

us determine the definitions which are required for recovering the potentially influencing definitions.

Listing 6 shows the pseudo code for the generation of a reduced reverse program. The notation *outgoing_dep_edge(n,i)* designates the $i^{th}$ outgoing use-definition and/or pseudo definition-predicate variable edge of node $n$. Similarly, *outgoing_def-recovering_edge(k,j)* designates the $j^{th}$ outgoing definition-recovering edge of node $k$. The function *follow_edge(n, e)* returns the node that is connected to node $n$ via edge $e$.

Given a dynamic slicing criterion $C = (X, L, I^q)$, and a global MVG, the RCGS algorithm first determines within the global MVG the node that corresponds to the definition of location $L$ at instruction $I$ (i.e., the node with respect to which we would like to take the slice) – line 1 of Listing 6. Let us designate the found node with $n$.

93

**Listing 6** Generate a Reduced Reverse Program

---

**Inputs**: A reverse program $RT$ and a global MVG

A dynamic slicing criterion $C = (X, L, I^q)$

**Output**: A reduced reverse program

**begin**

1 In global MVG, find node $n$ that corresponds to definition of $L$ at $I$
2 worklist = $\{n\}$
3 **repeat**
4    $n$ = head_of(worklist)
5    **for** $i = 1$ to $|$ outgoing_dep_edges($n$) $|$ **do**
6      $k$ = follow_edge($n$, outgoing_dep_edge($n,i$))
7      **if** $k \notin$ worklist **then**
8        worklist += $k$
9      **end if**
10      **if** $k \notin$ essential_defs **then**
11        essential_defs += $k$
12      **end if**
13      **for** $j = 1$ to $|$ outgoing_def-recovering_edges($k$) $|$ **do**
14        $m$ = follow_edge($k$, outgoing_def-recovering_edge($k,j$))
15        **if** $m \notin$ essential_defs **then**
16          essential_defs += $m$
17        **end if**
18      **end for**
19      worklist -= n
20    **end for**
21 **until** worklist = $\phi$
22 Pick from $T$ the RIGs which recover essential_defs
23 Update the target addresses of the remaining branches if necessary

**end**

---

As mentioned in Section 5.1.2, the use-definition edges and the pseudo definition-predicate variable edges in an MVG designate the data and control dependencies, respectively. Moreover, as explained in Section 7.1, the definitions that might influence $n$ are the definitions on which $n$ is either directly or indirectly data and/or control dependent. Therefore, the RCGS algorithm follows the use-definition and pseudo definition-predicate variable edges starting from $n$ and adds each newly visited node $k$ to the set of essential definitions as defined in Section 7.3 (lines 5, 6, 10 and 11). On the other hand, when each such node $k$ is added to the essential definitions set, the RCGS algorithm also finds the nodes that are connected to $k$ via definition-recovering edges (lines 13 and 14). The found nodes correspond to the definitions that are required for recovering $k$. Therefore, these nodes, if not already added, are also added

94

to the set of essential definitions (lines 15 and 16). After finding all essential definitions, we pick from the reverse program the RIGs that recover the essential definitions (line 22).

After the unnecessary RIGs are removed to generate the reduced reverse program, some instructions may shift in the reverse program. Therefore, in such a case, we also update the branch target addresses accordingly (line 23).

The following example illustrates how we generate a reduced reverse program by using a global MVG.

**Example 22** Figures 31(a) and 31(b) show an example program and the corresponding global MVG, respectively. For clarity, we do not show definition-recovering edges in Figure 31(b). Instead, definition-recovering relationships between the nodes are shown by the table in Figure 31(e). As in Figure 6, we use indices to refer to the instructions of the program under consideration. Suppose that we would like to take the slice with respect to $r_5$ at the instruction marked with index 9 in Figure 31(a). This instruction defines the value $r_5^1$. Therefore, we start with the node $r_5^1$ and follow the use-definition and pseudo definition-predicate variable edges while adding each visited node and its recovering definitions to the essential definitions set. The resulting nodes in the essential definitions set are shaded in Figure 31(b).

After the essential definitions and their corresponding RIGs are determined, the remaining task is to remove the rest of the RIGs from the reverse program to generate the reduced reverse program. The complete reverse program and the resulting reduced reverse program are shown in Figures 31(c) and 31(d), respectively. An instruction in the original program and the RIG which reverses that instruction are again marked with same index. For explanation purposes, we annotated each destroyed definition in Figure 31(b) with the RIG that recovers it. For instance, $RIG_9$ restores the initial value of $r_5$ which is named as $r_5^0$. Therefore, in Figure 31(b), we

**(a)**

```
        save    r2
i1      addi    r2, r1, 1    // r2 = r1 + 1
        save    r3
i2      mulli   r3, r2, 5    // r3 = r2 / 5
        save    r4
i3      mulli   r4, r1, 4    // r4 = r1 × 4
i4      lwzu    r4, 0(r4)    // r4 = mem(r4)
    L4: cmpi    r3, 64       // if r3 ≥ 64
        bge     L1           // goto L1
i5      rlwinm  r2, r3, -1   // r2 = r3 >> 1
        cmpi    r2, 0        // if r2 ≠ 0
        bne     L2           // goto L2
i6      add     r1, r3, r1   // r1 = r3 + r1
        b       L3           // goto L3
i7  L2: add     r4, r4, r3   // r4 = r4 + r3
i8  L3: addi    r3, r3, 1    // r3 = r3 + 1
        b       L4           // goto L4
        save    r5
i9  L1: addi    r5, r1, 9    // r5 = r1 + 9
```

**(b)**

(Corresponding MVG — multiversion graph)

**(c)**

```
RIG9            restore r5
RIG8 L1: subi   r3, r3, 1    // r3 = r3 - 1
         cmpwi  r2, 0        // if r2 ≠ 0
         beq    L5           // goto L5
RIG7     subi   r4, r4, r3   // r4 = r4 − r3
         b      L4           // goto L4
RIG6 L5: subi   r1, r1, r3   // r1 = r1 − r3
RIG5 L4: cmpwi  r3, 0        // if r3 ≠ 0
         bne    L2           // goto L2
         add    r2, r1, 1    // r2 = r1 + 1
         b      L3           // goto L3
     L2: subi   rt, r3, 1    // rt = r3 - 1
         rlwinm r2, rt, -1   // r2 = rt >> 1
     L3: cmpwi  r3, 0        // if r3 ≠ 0
         bne    L1           // goto L1
RIG4     mulli  r4, r1, 4    // r4 = r1 × 4
RIG3     restore r4
RIG2     restore r3
RIG1     restore r2
```

**(d)**

```
RIG8 L1: subi   r3, r3, 1    // r3 = r3 - 1
         cmpwi  r2, 0        // if r2 ≠ 0
         beq    L5           // goto L4
         b      L4
RIG6 L5: subi   r1, r1, r3   // r1 = r1 − r3
RIG5 L4: cmpwi  r3, 0        // if r3 ≠ 0
         bne    L2           // goto L2
         add    r2, r1, 1    // r2 = r1 + 1
         b      L3           // goto L3
     L2: subi   rt, r3, 1    // rt = r3 - 1
         rlwinm r2, rt, -1   // r2 = rt >> 1
     L3: cmpwi  r3, 0        // if r3 ≠ 0
         bne    L1           // goto L1
```

**(e)**

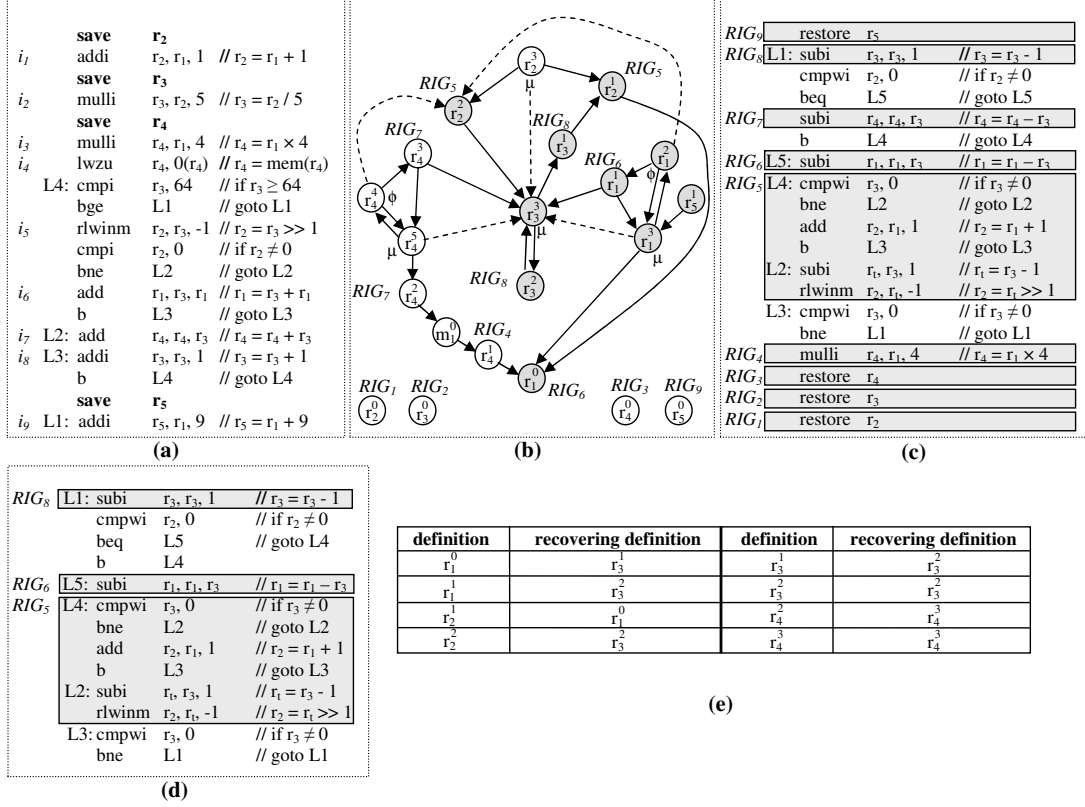| definition | recovering definition | definition | recovering definition |
|---|---|---|---|
| $r_1^0$ | $r_3^1$ | $r_3^1$ | $r_3^2$ |
| $r_1^1$ | $r_3^2$ | $r_3^2$ | $r_3^2$ |
| $r_2^1$ | $r_1^0$ | $r_4^2$ | $r_4^3$ |
| $r_2^2$ | $r_3^2$ | $r_4^3$ | $r_4^3$ |

**Figure 31:** (a) An example program. (b) Corresponding MVG. (c) The complete reverse program. (d) The reduced reverse program. (e) Table showing definition-recovering relationships.

annotated the node of $r_5^0$ with "$RIG_9$" (note that some nodes in Figure 31(b) do not have associated RIGs because these nodes either are not destroyed at all or do not represent real definitions but only pseudo definitions). As seen from Figures 31(c) and 31(d), the reduced reverse program includes only those RIGs that correspond to the essential definitions. □

## 7.5   Summary

Dynamic slicing support provides a programmer with the ability to reverse execute a program along a designated dynamic slice. In this way, the programmer deals with smaller portions of the whole program state to locate the bugs. In this chapter, we described how dynamic slicing support is added on top of the reverse code generation

algorithm. Basically, dynamic slicing is achieved by constructing a reduced reverse program from a complete reverse program. The reduced reverse program generated for a particular dynamic slice excludes the instructions that are not required to recover the state relevant to that slice. Therefore, the reduced reverse program, when executed, reveals the program state that is in the coverage of the chosen dynamic slice only.

In the next chapter, we present various experiments we performed to show the benefits of our instruction level reverse execution as well as our dynamic slicing technique.

# CHAPTER VIII

# PERFORMANCE EVALUATION

In this chapter, we present the results obtained by applying the techniques described in this thesis on a set of benchmark programs. We first describe the experimentation platform used for the measurements and explain the benchmark programs. Then, we show the results for full-scale reverse execution of the benchmarks. Finally, we illustrate the results for reverse execution over different dynamic slices. Throughout the experiments, our technique is quantitatively compared against the state-of-the-art techniques.

## 8.1   The Experimentation Platform

The target platform we chose to carry out measurements is an MBX860 evaluation board with a PowerPC (MPC860) processor and 4MB DRAM [38]. The reason we specifically chose the MBX860 board is that we wanted to explore the advantages of our technique in a real memory-restricted platform. However, our methodology explained in this thesis is also applicable to other platforms such as large general purpose computers.

In order to test instruction level reverse execution on a debugging session, we implemented a low-level debugger tool with a graphical user interface (GUI) which provides debugging capabilities such as breakpoint insertion, single stepping, register dispay and memory display (Figure 32). The debugger runs on a PC with Windows 2000. The PC is connected to the MBX860 board via a *Background Debug Mode* (*BDM*) interface [37]. We did not install any operating system on the MBX860 and
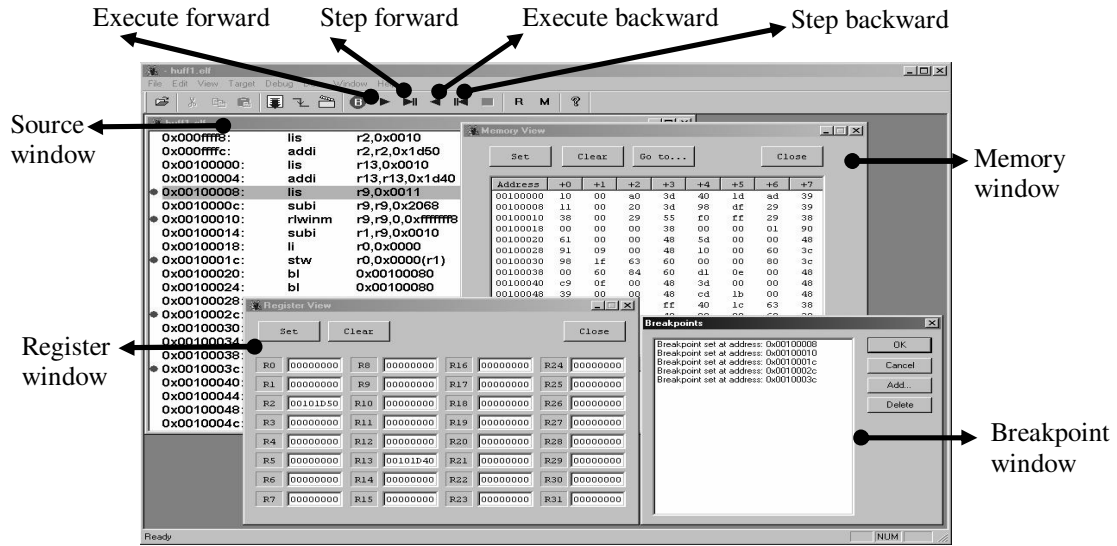
**Figure 32:** The GUI of the debugger tool.

ran the benchmarks directly on the processor. Therefore, the measurement results presented in this chapter do not include any operating system related overheads.

## 8.2 Benchmark Programs

The benchmark programs we used for our experimentation are selection sort (SSort), matrix multiply (MMult), Adaptive Differential Pulse Code Modulation (ADPCM) encoder from Media Bench [33] and Lempel Ziv Welch (LZW) code compression. These benchmarks are all integer benchmarks. The primary reason for selecting integer benchmarks is that the MPC860 processor does not have a floating point unit. Moreover, our methodology performs better with integer benchmarks because, as explained in Section 4.2, a floating point operation does not allow the use of the extract-from-use technique due to a possible precision loss, reducing the applicability of state regeneration.

SSort orders integer numbers which are input to SSort in an array. MMult multiplies two integer matrices which are input to MMult as arrays and writes the resulting matrix into another array. Finally, ADPCM and LZW read their inputs starting from

a location in memory and write the processed outputs back to another location in memory. The input data is written to memory prior to execution of ADPCM and LZW. SSort, MMult and ADPCM are implemented as a single function each, while LZW is composed of three functions. The main function of LZW calls the other two functions in a loop. One of the functions that is called from main includes a recursive call to itself inside a loop. ADPCM also reads and processes its input in a loop. On the other hand, SSort contains a two-level nested loop, and MMult contains a three-level nested loop.

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. [45]. Note that since we do not enforce any structural constraints on the assembly code that is input to the RCG algorithm, the input assembly code can be generated in any way even by an optimizing compiler. In our experiments, we compiled each benchmark using level-3 optimizations which include global common subexpression elimination, constant propagation, constant folding, dead code elimination, strength reduction, tail merging, spill-code reduction, loop memory-reference elimination and global register allocation. Level-3 optimization is the highest possible for GCC [23]. Thus, our experiments were performed on assembly optimized at the highest (most complex and aggressive) level possible for a typical compiler; thus, RCG and RCGS seem to be compatible even with highly aggressive compiler optimizations.

## 8.3    Results for Full-scale Reverse Execution

This section presents several performance evaluations of the RCG algorithm and compares the results against the results obtained from traditional reverse execution methods.

As explained in Chapter 2, none of the previous techniques provides instruction level reverse execution. Therefore, in order to compare the performance of the RCG

algorithm against the previous techniques, we had to expand the previous techniques to support instruction level reverse execution. Two of the best previous techniques that are expandable to support instruction level reverse execution without any forward execution are converted into either saving the modified processor state before each instruction – incremental state saving (ISS) – or saving the modified processor state before each destructive instruction (i.e., an instruction whose target operand is different than the source operands) – incremental state saving for destructive instructions (ISSDI). We used GCC level-3 optimized assembly code for applying ISS and ISSDI as well.

### 8.3.1   Comparison of Reverse Code Sizes

Table 1 shows the sizes of the compiled benchmarks and the reverses of the benchmarks for ISS, ISSDI and RCG. Note that since ISS and ISSDI do not actually generate a reverse program, the term "reverse code" as used in ISS and ISSDI refers to the instructions that recover the saved state in ISS and ISSDI.

The reverse code sizes obtained with RCG are approximately 1.18X to 2.19X larger than those that are obtained with ISS and ISSDI. This is because while ISS and ISSDI usually use a simple load instruction to restore a value in a register or a memory location, RCG uses a RIG that may be composed of multiple instructions. Another interesting result is that for SSort, MMult and LZW it turns out that each reverse code obtained by ISS has the same size as the equivalent reverse code obtained by ISSDI. This is because ISSDI simply removes some of the state restoring load instructions from the reverse code obtained by ISS and replaces them by equal number of arithmetic instructions that undo constructive reversible instructions (as described in Section 2.1.2). The rest of the instructions, on the other hand, are kept intact.

**Table 1:** Sizes of the original and the reverse codes.

| | code size (bytes) | reverse code size (bytes) | | |
|---|---|---|---|---|
| | | ISS | ISSDI | RCG |
| Selection sort | 152 | 180 | 180 | 284 |
| Matrix multiply | 204 | 232 | 232 | 508 |
| ADPCM encoder | 364 | 536 | 524 | 632 |
| LZW | 432 | 572 | 572 | 712 |

### 8.3.2 Comparison of Runtime Memory Usage

In this section, we illustrate the runtime memory usage results with ISS, ISSDI and RCG. Our goal is to show that the RCG algorithm provides reverse execution with much less runtime memory usage as compared to pure state saving approaches.

The tests are categorized according to various input data sizes to explore how runtime memory requirements grow with the problem size. For SSort, the number of elements to be sorted were chosen as 100, 1000 and 10000. For MMult, we used 4x4, 40x40 and 400x400 matrices. We fed ADPCM with 32KB, 64KB and 128KB input data sizes. Finally, the input data sizes for LZW were varied between 1KB, 4KB and 16KB.

In this experiment, we ran each benchmark instrumented with state saving instructions according to ISS, ISSDI and RCG from the beginning forward until the end. Then, we measured the total runtime memory used for state saving. State saving was applied using circular buffers which never overflow so that we could measure total memory usage even though the memory requirements with ISS and ISSDI in many cases were above the available amount on the MBX860. The results are shown in Table 2. As seen in the table, when data sizes are increased, the memory requirements with ISS and ISSDI quickly exceed the 4MB of available memory on the MBX860, while RCG still provides feasible memory usage except for the case when the input

size of SSort is 10000. However, even in this case, the runtime memory requirement with RCG is 82X and 55X smaller than the runtime memory requirements with ISS and ISSDI, respectively. In general, RCG achieves from 2.5X to 2206X and from 2X to 1404X reduction in runtime memory usage as compared to ISS and ISSDI, respectively.

**Table 2:** Runtime memory requirements for state saving.

|  | ISS (KB) | ISSDI (KB) | RCG (KB) | ISS / RCG | ISSDI / RCG |
|---|---|---|---|---|---|
| Selection sort (100 inputs) | 68.2 | 46.9 | 7.5 | 9X | 6.3X |
| Selection sort (1000 inputs) | 6032 | 4065 | 151 | 40X | 27X |
| Selection sort (10000 inputs) | 593389 | 397913 | 7237 | 82X | 55X |
| Matrix multiply (4x4) | 3.6 | 2.35 | 0.17 | 21X | 14X |
| Matrix multiply (40x40) | 2820 | 1801 | 12.6 | 224X | 143X |
| Matrix multiply (400x400) | 2756883 | 1755006 | 1250 | 2206X | 1404X |
| ADPCM (32KB input data) | 1544 | 1192 | 616 | 2.5X | 2X |
| ADPCM (64KB input data) | 3088 | 2384 | 1232 | 2.5X | 2X |
| ADPCM (128KB input data) | 6175 | 4767 | 2464 | 2.5X | 2X |
| LZW (1KB input data) | 5630 | 3425 | 98.4 | 57X | 35X |
| LZW (4KB input data) | 64970 | 39163 | 351 | 185X | 112X |
| LZW (16KB input data) | 784336 | 471140 | 1331 | 589X | 354X |

### 8.3.3 Comparison of Execution Times

In this section, we provide forward and reverse execution times of the benchmarks with ISS, ISSDI and RCG. For the execution time measurements, we used the decrementer counter of the PowerPC 860 processor (the PowerPC 860 provides a decrementer counter which can be programmed to decrement at 2.5MHz on the MBX860; therefore, one tick of the decrementer corresponds to 0.4 microseconds). We measured the forward execution times both for the original benchmarks and for the benchmarks that are instrumented with state saving instructions. In this way, we could also measure the forward execution time overheads that are caused by ISS, ISSDI and RCG. To measure the forward execution times, each benchmark was run from the

**Table 3:** Forward execution time measurements of the original programs.

| Benchmark | Raw Execution Time (decrementer ticks) | Raw Execution Time (seconds) |
|---|---|---|
| Selection sort (100 inputs) | 21187 | 0.008475 |
| Selection sort (1000 inputs) | 2000202 | 0.800081 |
| Selection sort (10000 inputs) | 198539130 | 79.41565 |
| Matrix multiply (4x4) | 650 | 0.00026 |
| Matrix multiply (40x40) | 472044 | 0.188818 |
| Matrix multiply (400x400) | 457183831 | 182.8735 |
| ADPCM (32KB input data) | 378294 | 0.151318 |
| ADPCM (64KB input data) | 751280 | 0.300512 |
| ADPCM (128KB input data) | 1496649 | 0.59866 |
| LZW (1KB input data) | 1380413 | 0.552165 |
| LZW (4KB input data) | 16063096 | 6.425238 |
| LZW (16KB input data) | 194451339 | 77.78054 |

**Table 4:** Execution time measurements of the instrumented and reverse programs.

| | Benchmark | ISS | ISSDI | RCG | ISS/ RCG | ISSDI/ RCG |
|---|---|---|---|---|---|---|
| Instrumented fwd. exec. time (decrementer ticks) | Selection sort (100 inputs) | 42984 | 38496 | 31113 | 1.38X | 1.24X |
| | Selection sort (1000 inputs) | 3979802 | 3595213 | 2841029 | 1.40X | 1.27X |
| | Selection sort (10000 inputs) | 394063091 | 356208073 | 280677488 | 1.40X | 1.27X |
| | Matrix multiply (4x4) | 1432 | 1197 | 708 | 2.02X | 1.70X |
| | Matrix multiply (40x40) | 1092872 | 895703 | 476243 | 2.29X | 1.88X |
| | Matrix multiply (400x400) | 1064415269 | 870539981 | 458691637 | 2.32X | 1.90X |
| | ADPCM (32KB input data) | 805972 | 737720 | 616101 | 1.31X | 1.20X |
| | ADPCM (64KB input data) | 1611572 | 1475276 | 1232110 | 1.31X | 1.20X |
| | ADPCM (128KB input data) | 3223166 | 2950562 | 2464232 | 1.31X | 1.20X |
| | LZW (1KB input data) | 3126206 | 2699287 | 2054657 | 1.52X | 1.31X |
| | LZW (4KB input data) | 36319691 | 31942838 | 23813230 | 1.52X | 1.34X |
| | LZW (16KB input data) | 439424024 | 378614957 | 288077045 | 1.53X | 1.31X |
| Reverse execution time (decrementer ticks) | Selection sort (100 inputs) | 28724 | 27137 | 36719 | 0.78X | 0.74X |
| | Selection sort (1000 inputs) | - | - | 3516414 | - | - |
| | Selection sort (10000 inputs) | - | - | - | - | - |
| | Matrix multiply (4x4) | 880 | 784 | 1325 | 0.66X | 0.60X |
| | Matrix multiply (40x40) | 660189 | 578556 | 1088827 | 0.61X | 0.53X |
| | Matrix multiply (400x400) | - | - | 1070219421 | - | - |
| | ADPCM (32KB input data) | 656702 | 628958 | 765036 | 0.86X | 0.82X |
| | ADPCM (64KB input data) | - | 1257770 | 1528807 | - | 0.82X |
| | ADPCM (128KB input data) | - | - | 3057176 | - | - |
| | LZW (1KB input data) | - | - | 2619106 | - | - |
| | LZW (4KB input data) | - | - | 30596864 | - | - |
| | LZW (16KB input data) | - | - | 371045637 | - | - |

beginning forward until the end, and to measure the reverse execution times, each benchmark was run from the end backward until the beginning.

Table 3 depicts the forward execution time results of the original benchmarks (i.e., the benchmarks that are not instrumented by state saving instructions). Table 4, on the other hand, shows the forward execution times of the instrumented benchmarks and the corresponding reverse execution times. The dashes in Table 4 correspond to measurements in which we ran out of memory on the MBX860. As shown in Table 4 (by a dash "-"), in most cases ISS and ISSDI do not let us reverse execute the whole program because the limited memory on the MBX860 can hold only a small fraction of the required state.

As seen in Tables 3 and 4, the execution times increase linearly for ADPCM, while they increase exponentially for SSort, MMult and LZW. This is in accordance with the loop structures in the benchmarks. Since instructions within the innermost loops are the most frequently executed instructions, the execution times of the benchmarks increase almost linearly with the increase in the number of times the innermost loops are executed. Since SSort, MMult and LZW include nested loops, we observe exponential increase in the measured execution times when the input data sizes are increased.

The slow down in reverse execution with RCG as compared to ISS and ISSDI is between 1.16X and 1.89X. This slow down is a direct consequence of larger reverse programs that are generated by RCG as compared to ISS and ISSDI. This is the only penalty we have to pay as we gain much from runtime memory as illustrated in Table 2.
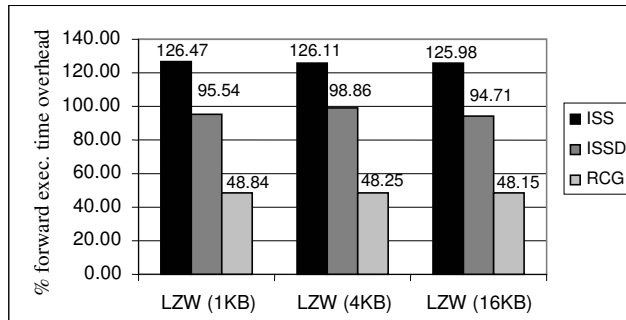
However, since reverse executions are usually followed by forward executions in cyclic debugging, the time loss during reverse execution can typically be compensated by the reduced forward execution times of the programs with RCG. Since ISS and ISSDI instrument the programs with many more state saving instructions than
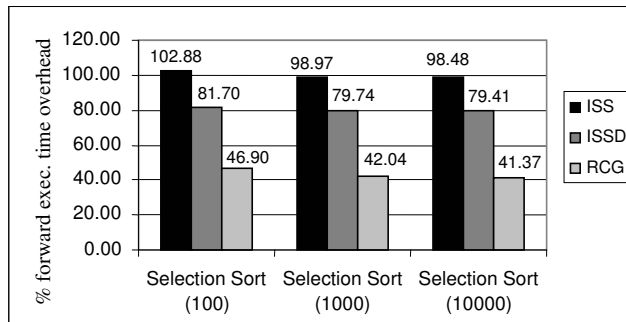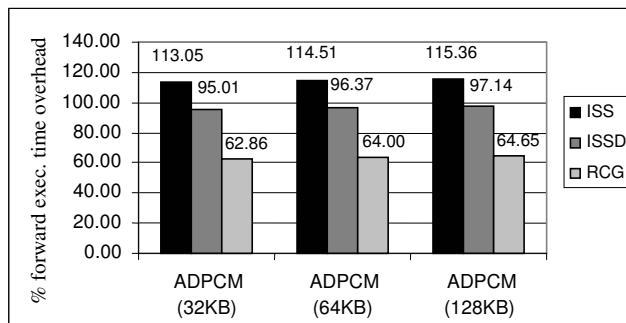
(a) Selection sort
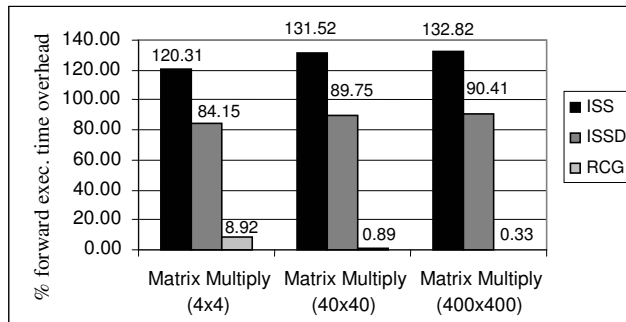


(b) Matrix multiply



(c) LZW



(d) ADPCM encoder

**Figure 33:** Execution time overhead results of ISS, ISSDI and RCG.

RCG, RCG achieves faster forward executions than those that are achieved by ISS and ISSDI. The speedup in forward execution with RCG over ISS and ISSDI ranges between 1.2X and 2.32X. This result is also seen by the forward execution time overheads shown in Figure 33. The forward execution time overhead indicates the percent ratio of the increase in the forward execution time (due to code instrumentation) over the execution time of the original (uninstrumented) code. RCG achieves from 1.5X to 403X reduction in forward execution time overheads as compared to ISS and ISSDI.

The last RCG measurement compares debugging via reverse execution with debugging via program re-execution. In this measurement, we executed a 400x400 matrix multiply from the beginning (end) to various intermediate program points in the forward (backward) direction and measured the elapsed times. The intermediate program points correspond to the beginning of the outermost loop of matrix multiply at different iteration instances of this loop. For 400x400 input matrices, the outermost loop executes 400 times in total.

The results are shown in Figure 34. Suppose just as the 400th outermost loop (the right-hand-side of Figure 34) ends, a bug is noticed. Suppose further that the bug source is suspected to be in the 300th loop iteration. Figure 34 shows that while it would take 107 seconds to reverse execute back to iteration 300 using RCG compiled code, it would take 137 seconds to forward execute uninstrumented code (the original code) from loop iteration zero forward to loop iteration 300. In fact, once at the 400th iteration of the outermost loop, for any loop iteration greater than 280 it is faster to reverse execute to the target loop iteration than to forward execute from loop iteration zero to the target loop iteration. In short, Figure 34 empirically demonstrates that for a fairly large set of potential bugs localized close to the current iteration point (in the case of Figure 34, the iteration point is the 400th iteration of the outermost loop), it is faster to use RCG to go to the bug than to re-execute the original code from the beginning.

**Figure 34:** The elapsed forward/reverse execution times versus various program points in 400x400 matrix multiply.

One option to speed up the program re-execute approach might be to take periodic checkpoints of the whole program state so that the programmer can restart the program from the nearest checkpoint instead of the beginning of the program. Then, he/she can forward execute from that point on to reach the target point. However, obviously, checkpointing of the whole program state requires more memory than incremental state saving if it is performed frequently enough to provide fast state restoration. For example, assume that we take periodic checkpoints of 400x400 matrix multiply every 100ms to provide a worst case reverse execution time of 100ms. To take an absolute checkpoint, typically, we at least need to save the value of every element in the result matrix. Since each element is a 4-byte integer, each checkpoint requires at least 625KB of memory. With the total runtime of 183 seconds, 400x400 matrix multiply requires 1830 checkpoints and thus approximately 1.1GB of runtime memory. However, RCG requires only 1250KB as shown in Table 2. Therefore, once the target point is reached, a checkpointing approach cannot achieve the speed of our technique without sacrificing large memory space.

## 8.4 Results for Reverse Execution Along a Dynamic Slice

This section presents the measurements we performed to test our technique which provides reverse execution along a dynamic slice. We first show the advantage of dynamic slicing support in terms of speed improvement during reverse execution. Then, we show the efficiency of the RCGS algorithm in terms of runtime memory requirements.

### 8.4.1 Reverse Execution Time Measurement

We first compare the overall execution time of full-scale reverse execution against the overall execution time of reverse execution over a dynamic slice. In this experiment, matrix multiply was performed over two 4x4 matrices and selection sort was performed over an array of 10 integers. ADPCM and LZW, on the other hand, were run over 128KB input data. We experimented over three different dynamic slices for each benchmark. For matrix multiply and ADPCM, we took the slices for two different registers, while for selection sort and LZW, each slice was taken for a single register. The slices for LZW were taken before the midpoint in execution so that we did not allow the whole memory on the MBX860 to be consumed by state saving. Again, the execution time measurements were performed via the decrementer counter of the PowerPC 860 processor in a nonintrusive way.

In this experiment, we reverse executed each benchmark from the end of each slice until the beginning of each slice first by using RCG and then by using RCGS.

Figure 35(a) shows the results for the four benchmark programs. The average speedups with RCGS compared to RCG for matrix multiply, selection sort, ADPCM and LZW are 35X, 2X, 1.3X and 2.7X, respectively. The reason matrix multiply gives much larger speedup than the other benchmarks is that RCGS can remove many RIGs from the nested loop in reverse of matrix multiply, while it can remove

fewer RIGs from the loops in the reverses of the other benchmarks. The average code size reduction from the complete reverse program to the reduced reverse program is the largest for matrix multiply with a factor of 6.6X, while the code size reduction factors for selection sort, ADPCM and LZW are 3.1X, 1.4X and 1.5X, respectively.

In order to better evaluate the advantage of reverse execution along a dynamic slice in terms of reverse execution time, we performed another set of measurements. We increased the input data sizes for matrix multiply and selection sort to increase the running time of these benchmarks. We experimented with the same three slices for each benchmark and took the average of the reverse execution times. The results are shown in Figures 35(b) and 35(c). For instance, with 400x400 matrix multiply, the full-scale reverse execution takes 4.5 minutes on average, while the reverse execution along a dynamic slice takes only 141 microseconds on average, a 1,928,500X reduction.

While the average reverse execution time of matrix multiply along a dynamic slice is almost kept constant with increased input size, the average reverse execution time of selection sort along a dynamic slice increases by a factor of $y^2$ when the input size is increased by $y$ times. The reason for this behavior can be explained as follows.

First, note that the execution time of the reverse of the three-level nested loop inside the full reverse matrix multiply code dominates the execution time of the rest of the reverse matrix multiply code because the execution time of the rest of the code does not grow with increasing input size. Second, when the dynamic slice is extracted, many of the RIGs that are removed from the reverse matrix multiply code are from within the three-level nested loop. Therefore, the remaining RIGs in the reverse matrix multiply code execute in almost constant time.

On the other hand, most of the RIGs are kept within the inner loop of the reverse of selection sort even after the dynamic slice is extracted. Since this inner loop executes $n^2$ times with $n$ being the number of integers to be sorted, we see an increase in the
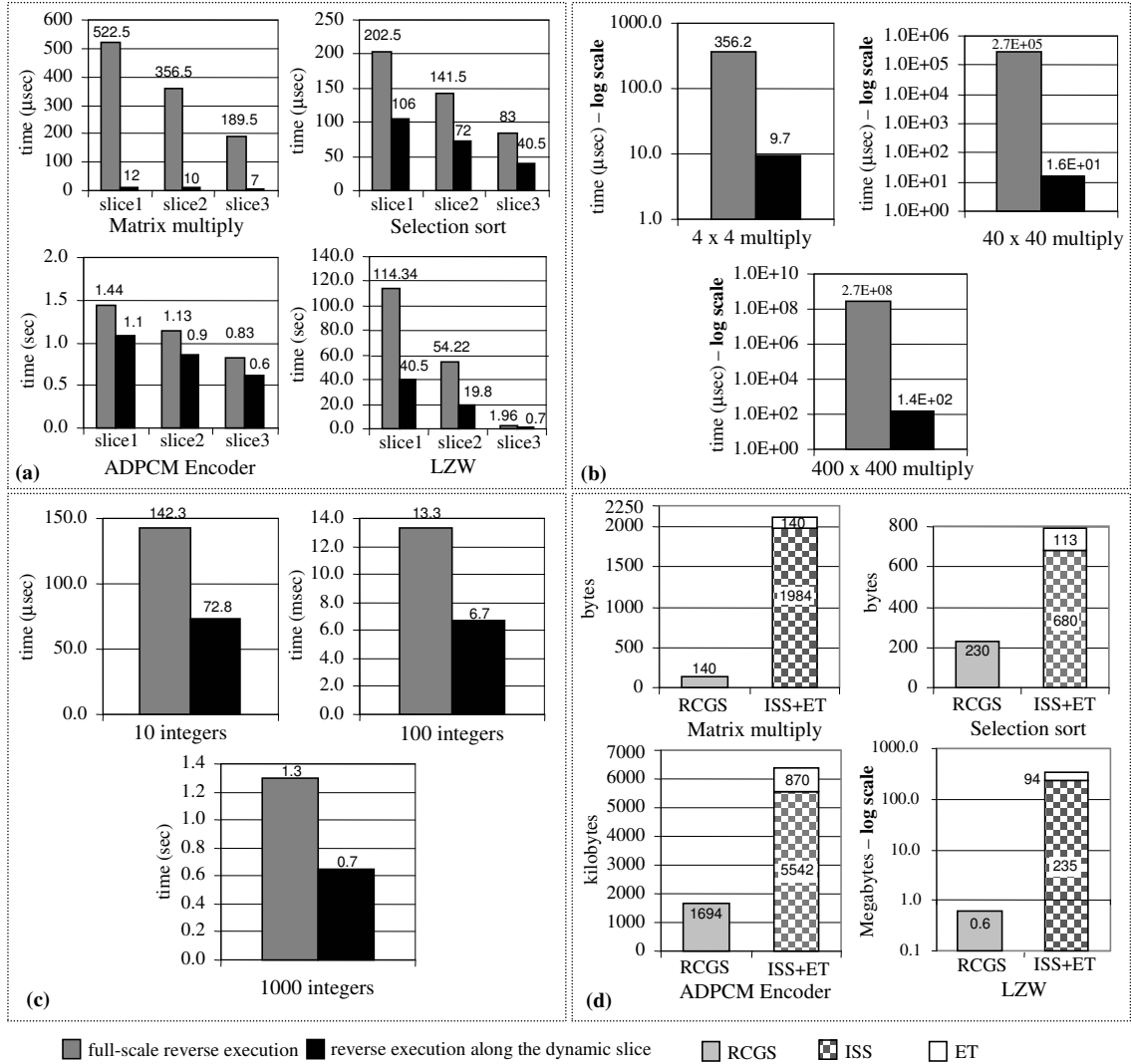
110

**Figure 35:** (a) Reverse execution time comparison. (b) Reverse execution time comparison of matrix multiply with different matrix sizes. (c) Reverse execution time comparison of selection sort with different input array sizes. (d) Runtime memory requirement comparison.

111

reverse execution time of selection sort by a factor of $y^2$ when $n$ is increased by $y$ times.

### 8.4.2 Measurement of Runtime Memory Usage

The RCGS algorithm provides important debugging support by implementing reverse execution along a dynamic slice with little runtime trace information. To show this, we implemented a reverse execution method with dynamic slicing support by using an incremental state saving technique (ISS) together with one of the best traditional dynamic slicing techniques which uses an execution trajectory (ET) [4]. We compared the runtime memory requirement of RCGS against the memory requirement of ISS and ET.

Figure 35(d) shows the results for the four benchmark programs. In this experiment, the input sizes of the benchmarks were chosen to be the same as the sizes in our first dynamic slicing experiment explained in Section 8.4.1. The results indicate that compared to ISS+ET, RCGS achieves approximately 15.2X, 3.4X, 3.8X and 548X reduction in memory overheads for matrix multiply, selection sort, ADPCM and LZW, respectively. Due to the loop effects explained in Section 8.4.1, these figures become even larger with increasing input data sizes. For instance, for 400x400 matrix multiply, while ISS+ET requires 1.37GB of memory, RCGS requires only 626KB. For selection sort with 1000 inputs, while ISS+ET requires 2.5MB of memory, the memory requirement of RCGS is only 98KB.

# CHAPTER IX

# CONCLUSION

Executing a program repeatedly is an effective debugging method applied by most programmers. However, every time a program is restarted, parts of the program that have already executed without error have to be re-executed unnecessarily. The unnecessary re-execution of these program parts constitute a significant portion of the debugging time. Even worse, restarting programs that run for very long time periods is simply impractical.

Reverse execution cuts down the time spent for repetitive or cyclic debugging by localizing program re-executions around the bugs in a program. When a bug location is missed by executing a program too far, the program state at a point before the bug location can be restored by reverse execution and the program can be re-executed from that point on without having to restart the whole program.

Conventional techniques rely heavily on saving processor state before the state is destroyed. However, state saving causes significant memory and time overheads during execution of a program. In an effort to reduce memory and time overheads caused by state saving, the state saving frequency can be reduced. However, reducing the state saving frequency increases the distance between the point where the program is stopped and the closest point from where the program can be restarted, which effectively reduces the benefit of reverse execution.

In this thesis, a new reverse execution methodology for programs has been 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, as a new feature, the methodology provides instruction by instruction reverse execution at the assembly instruction level without ever requiring any forward execution of the program under consideration. In this way, a program can be run backwards to a state as close as one assembly instruction before the current state, which is very useful for debugging programs written in assembly or programs whose assembly has been aggressively optimized by a compiler.

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

This thesis also introduced a new approach to reverse execution along a dynamic slice. Specifically, the introduced RCGS approach provides an instruction level reverse execution that visits only the dynamic slice instructions and that skips recovering program state unrelated to the dynamic slice. This results in a fast reverse execution at the assembly instruction level. Moreover, the approach does not require an execution trajectory to extract a dynamic slice from a program. Instead, the necessary runtime information is mainly reconstructed during reverse execution by the control flow predicates in the reverse program. This property coupled with the regeneration of runtime values on the fly makes the techniques embodied in RCGS as presented in this thesis very memory efficient.

In conclusion, this thesis achieves for the first time known to the author reverse execution of an assembly program via execution of a "reverse program" generated from the original assembly program input. Dramatic reductions (up to several orders of magnitude) in runtime memory requirements are achieved compared to previous

approaches, all of which end up relying heavily on state saving to memory since they do not have access to any "reverse program." In short, standard compiler techniques – plus some new, novel techniques specific to reversing the effects of assembly instructions – are used to generate the reverse program. In addition, the reverse program can be pruned to include only information related to a dynamic slice, resulting in up to six orders of magnitude reduction in reverse execution time (as compared to the time spent to reverse execute the entire program). Finally, as compared to one of the best traditional dynamic slicing techniques, reverse execution along a dynamic slice using a pruned reverse program results in up to two orders of magnitude reduction in runtime memory overhead on candidate benchmarks.

# APPENDIX A

# HANDLING EFFECTS OF INDIRECTLY

# MODIFIED LOCATIONS

We have already stated in Section 3.1 that a reverse program $RT$ generated for a program $T$ recovers only memory and register values that are directly modified by the instructions of $T$. This appendix explains how the effects of indirectly modified memory and register values can be handled to ensure correct operation during a debugging session.

A value is *indirectly* modified if the underlying processor updates the value implicitly as a result of an executed instruction. In other words, an indirectly modified location does not appear as an operand in the encoding of an instruction. Indirectly modified locations are usually status bits that flag certain conditions such as overflows. Consider the following example.

**Example 23** Many processors usually modify a condition bit in a specific register after executing a "compare" instruction. This bit determines whether a subsequent conditional branch instruction will be taken or not. A condition bit may not necessarily be apparent in the encoding of a compare instruction but may be updated by the processor implicitly after a compare instruction executes. Thus, such a condition bit is an example of an indirectly modified location. □

Let us designate by $I$ the set of instructions of a processor $P$. 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 $E'$ ($E' \subset I$ and $E' = I - E$), on the other hand, are affected by indirectly modified memory and/or register values.

**Example 24** Consider an ordinary "add" instruction. The outcome of an ordinary "add" 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 can only be modified in a direct fashion via other instructions in the program. Therefore, an ordinary "add" instruction is an element of $E$. On the other hand, consider an "extended add" instruction whose result is the sum of its source operands plus a carry bit which might have been indirectly modified by a prior instruction. Therefore, an "extended add" instruction is an element of $E'$. □

**Table 5:** Distribution of the instructions in $E$ and $E'$ for the MPC860.

| Instructions in $E'$ affected by | Number | Percentage | Comments |
|---|---|---|---|
| condition bit | 4 | 2.80 | determines if a branch will be taken or not |
| carry bit | 6 | 4.20 | affects the result of extended addition/subtraction instructions |
| overflow bit | 4 | 2.80 | branches can be taken based on an overflow bit |
| status register | 2 | 1.40 | is indirectly modified and can affect a general purpose register by a direct copy |
| timer | 1 | 0.70 | is indirectly modified and can affect a general purpose register by a direct copy |
| reserve bit | 1 | 0.70 | affects the outcome of a store conditional instruction which is used to implement a synchronization primitive |
| **total in $E'$** | 18 | 12.59 | |
| **total in $E$** | 125 | 87.41 | |
| **total in $I$** | 143 | 100.00 | |

Table 5 shows an estimated distribution of the instructions in $E$ and $E'$ for the MPC860 processor. Also, Table 5 categorizes the instructions in $E'$ according to which indirectly modified location affects the instructions in $E'$. As seen in the table, 87.41%

117

of the instructions in MPC860 instruction set are inside $E$ and can be safely reversed by the reverse code. On the other hand, 12.59% of the MPC860 instructions require specific treatment by use of the re-execute approach or the state saving approach described in this appendix.

**Handling the Effects of Instruction in $E'$**

The only potential problem due to not recovering indirectly modified values via a reverse program is that an instruction in $E'$ may not execute consistently after reverse execution. Thus, if we can always make the instructions in $E'$ execute consistently after reverse execution, the execution will be always correct.

Let us assume that the outcome of an instruction $\alpha \in E'$ depends on the value $V$ of an indirectly modified register or a memory location. Also, assume that an instruction, say $\beta$, computes $V$ indirectly. We propose two approaches that can be used to provide consistent execution of $\alpha$ even though $V$ is not recovered via a reverse program. The first approach is to re-execute $\beta$ just before every time $\alpha$ is executed. Thus, even if $V$ is not recovered during reverse execution, re-execution of $\beta$ will restore $V$ before $\alpha$ executes. The second approach is to save the state (i.e., the state $\alpha$ modifies) before $\alpha$ is reverse executed and to restore the saved state after $\alpha$ is forward executed subsequently. Thus, the restoration of the saved state ensures that the state is kept consistent after execution of $\alpha$. Note that this state saving is repeated for every instruction that may be affected by an indirectly modified location. Let us illustrate these two approaches by the following two examples.

**Example 25** *The re-execute approach:* Consider the following instruction sequence (the numbers in 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 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 instruction level reverse execution. □

**Example 26** *The state saving approach*: Consider the following instruction sequence (again, the numbers in parentheses show the program points):

(1)

*add*  $r_9, r_2, r_3$

(2)

*adde* $r_{12}, r_{11}, r_{10}$

(3)

*add*  $r_{13}, r_1, r_6$

(4)

...

The outcome of the "extended add" instruction (indicated by mnemonic "adde") depends on the value of the carry bit set by the previous "add" instruction. Assume that during the initial forward execution, the first "add" instruction sets the carry bit to '1'; thus, the "adde" instruction uses the carry bit of value '1'. Also, assume that the "add" instruction after the "adde" instruction clears the carry bit during the initial forward execution. Therefore, if the programmer reverse executes the program from point (4) to point (2) and then forward executes the program from point (2) back to point (3), the "adde" instruction will compute an incorrect result into $r_{12}$ because "adde" will use the carry bit of value '0' this time.

To prevent the above situation, the debugger saves the value of $r_{12}$ just before the programmer reverse executes the "adde" instruction. When the programmer forward executes the program from point (2) to point (3) in the above scenario, the saved value of $r_{12}$ is restored ensuring the consistency of the value of $r_{12}$. $\square$

The re-execute approach requires that the values on which $\beta$ depends are not overwritten at an intermediate point between $\beta$ and $\alpha$. This is to ensure that $\beta$ calculates $V$ correctly when $\beta$ is re-executed. Therefore, the re-execute approach might be costly in terms of analysis time as the re-execute approach requires a separate data-flow analysis. On the other hand, the state saving approach explained above does not require a data-flow analysis and thus is potentially faster than the re-execute approach. However, the state saving approach results in more runtime memory usage to save the state before $\alpha$ is reverse executed. In summary, the re-execute approach can always be applied (since the direct operands of the instruction(s) to be re-executed can always be obtained, e.g., via state saving), and thus any and all indirectly modified locations can always be properly restored.

# REFERENCES

[1] ADL-TABATABAI, A. and GROSS, T., "Detection and recovery of endangered variables caused by instruction scheduling," in *Proceedings of the ACM SIG-PLAN'93 Conference on Programming Language Design and Implementation*, pp. 13–25, 1993.

[2] AGRAWAL, H., DEMILLO, R. A., and SPAFFORD, E. H., "Dynamic slicing in the presence of unconstrained pointers," in *Fourth ACM Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 60–73, 1991.

[3] AGRAWAL, H., DEMILLO, R. A., and SPAFFORD, E. H., "An execution back-tracking approach to debugging," *IEEE Software*, vol. 8, pp. 21–26, May 1991.

[4] AGRAWAL, H. and HORGAN, J., "Dynamic program slicing," *SIGPLAN Notices*, vol. 25, pp. 246–256, June 1990.

[5] AHO, A. V., SETHI, R., and ULLMAN, J. D., *Compilers: Principles, Techniques, and Tools*. MA: Addison-Wesley, 1986.

[6] ALPERN, B., WEGMAN, M. N., and ZADECK, F. K., "Detecting equalities of variables in programs," in *15th Annual ACM Symposium on Principles of Programming Languages*, pp. 1–11, January 1988.

[7] BACON, D. F. and GOLDSTEIN, S. C., "Hardware-assisted replay of multiprocessor programs," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, vol. 26, pp. 194–206, December 1991.

[8] BERGERETTI, J.-F. and CARRE, B. A., "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 37–61, 1985.

[9] BESZEDES, A., GERGELY, T., SZABO, Z. M., CSIRIK, J., and GYIMOTHY, T., "Dynamic slicing method for maintenance of large C programs," in *5th European Conference on Software Maintenance and Reengineering*, pp. 105–113, 2001.

[10] BHARGAVA, B. K., "Concurrency control in database systems," *Knowledge and Data Engineering*, vol. 11, no. 1, pp. 3–16, 1999.

[11] BINKLEY, D. W. and GALLAGHER, K. B., "Program slicing," *Advances in Computers*, vol. 43, pp. 1–50, 1996.

[12] BIRCH, M. R., BORONI, C. M., GOOSEY, F. W., PATTON, S. D., POOLE, D. K., PRATT, C. M., and ROSS, R. J., "Dynalab," *ACM SIGCSE Bulletin*, vol. 27, pp. 29–33, March 1995.

[13] BISWAS, B. and MALL, R., "Reverse execution of programs," *ACM SIGPLAN Notices*, vol. 34, pp. 61–69, April 1999.

[14] BOOTH, S. P. and JONES, S. B., "Walk backwards to happiness - debugging by time travel," in *Automated and Algorithmic Debugging*, pp. 171–183, 1997.

[15] CAROTHERS, C., PERUMALLA, K., and FUJIMOTO, R., "Efficient optimistic parallel simulations using reverse computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, pp. 224–253, July 1999.

[16] CHANDY, K. M. and RAMAMOORTHY, C. V., "Rollback and recovery strategies for computer programs," *IEEE Transactions on Computers*, vol. 21, pp. 546–556, June 1972.

[17] COOK, J. J., "Reverse execution of java bytecode," *The Computer Journal*, vol. 45, pp. 608–619, May 2002.

[18] CRESCENZI, P., DEMETRESCU, C., FINOCCHI, I., and PETRESCHI, R., "Reversible execution and visualization of programs with Leonardo," *Journal of Visual Languages and Computing (JVLC)*, vol. 11, pp. 125–150, April 2000.

[19] FELDMAN, S. I. and BROWN, C. B., "IGOR: A system for program debugging via reversible execution," in *Workshop on Parallel and Distributed Debugging*, pp. 112–123, 1988.

[20] FLEISCHMANN, J. and WILSEY, P. A., "Comparative analysis of periodic state saving techniques in time warp simulators," in *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, pp. 50–58, 1995.

[21] FLOYD, R. W., "Nondeterministic algorithms," *Journal of the ACM*, vol. 14, pp. 636–644, October 1967.

[22] FUJIMOTO, R. M., "Time warp on a shared memory multiprocessor," *Transactions of the Society for Computer Simulation International*, vol. 6, pp. 211–239, July 1989.

[23] GNU, *GCC Home Page*.
http://gcc.gnu.org.

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

[25] GRAY, J. and REUTER, A., *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[26] HARMAN, M. and DANICIC, S., "A new approach to program slicing," in *7th International Software Quality Week*, May 1994.

[27] HARMAN, M. and GALLAGHER, K. B., "Special issue on program slicing," *Journal of Information and Software Technology*, vol. 40, November/December 1998.

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

[29] HORWITZ, S., REPS, T., and BINKLEY, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[30] JEFFERSON, D. A., "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[31] KOREL, B. and LASKI, J. W., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[32] KOREL, B. and YALAMANCHILI, S., "Forward computation of dynamic program slices," in *International Symposium on Software Testing and Analysis (ISSTA)*, 1994.

[33] LEE, C., POTKONJAK, M., and MANGIONE-SMITH, W. H., "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *International Symposium on Microarchitecture*, pp. 330–335, 1997.

[34] LEE, Y.-H. and SHIN, K. G., "Design and evaluation of a fault tolerant multiprocessor using hardware recovery blocks," *IEEE Transactions on Computers*, vol. 33, pp. 113–124, February 1984.

[35] MARUYAMA, K. and TERADA, M., "One-pass pseudo reverse execution of C programs," *IPSJ Transactions on Programming*, vol. 41, no. SIG09 - 002, 2000.

[36] MILLER, B. P. and CHOI, J., "A mechanism for efficient debugging of parallel programs," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 135–144, 1988.

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

[38] Motorola Inc., *MBX860 Datasheet*, 2002. https://mcg.motorola.com/us/ds/pdf/ds0134.pdf.

[39] MUCHNICK, S. S., *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann, 1997.

[40] NETZER, R. H. B. and WEAVER, M. H., "Optimal tracing and incremental reexecution for debugging long-running programs," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 313–325, 1994.

[41] OTTENSTEIN, K. and OTTENSTEIN, L., "The program dependence graph in a software development environment," in *ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pp. 177–184, April 1984.

[42] PAN, D. Z. and LINTON, M. A., "Supporting reverse execution of parallel programs," in *Workshop on Parallel and Distributed Debugging*, pp. 124–129, 1988.

[43] SOSIC, R., "History cache: Hardware support for reverse execution," *Computer Architecture News*, vol. 22, pp. 11–18, December 1994.

[44] STOLPER, S., "Questions and answers about the Mars Pathfinder," October 1997.
http://www.quest.arc.nasa.gov/mars/ask/about-mars-path.

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

[46] TIP, F., "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, September 1995.

[47] WEISER, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352–357, July 1984.

[48] WEST, D. and PANESAR, K. S., "Automatic incremental state saving," in *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, pp. 78–85, 1996.

[49] WISMULLER, R., "Debugging of globally optimized programs using data flow analysis," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 278–289, 1994.

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

[51] ZHANG, X., GUPTA, R., and ZHANG, Y., "Precise dynamic slicing algorithms," in *IEEE/ACM International Conference on Software Engineering*, pp. 319–329, 2003.