A Fast Assembly Level Reverse Execution Method via Dynamic Slicing

Tankut Akgul*, Vincent J. Mooney III*,+ and Santosh Pande+
School of Electrical and Computer Engineering*, College of Computing+
Georgia Institute of Technology, Atlanta, GA 30332, USA
{tankut, mooney}@ece.gatech.edu, santosh@cc.gatech.edu

Abstract

One of the most time consuming parts of debugging is trying to locate a bug. In this context, there are two powerful debugging aids which shorten debug time considerably: reverse execution and dynamic slicing. Reverse execution eliminates the need for repetitive program restarts every time a bug location is missed. Dynamic slicing, on the other hand, isolates code parts that influence an erroneous variable at a program point. In this paper, we present an approach which provides assembly level reverse execution along a dynamic slice. In this way, a programmer not only can find the instructions relevant to a bug, but also can obtain runtime values of variables in a dynamic slice while traversing the slice backwards in execution history.

Reverse execution along a dynamic slice skips recovering unnecessary program state; therefore, it is potentially faster than full-scale reverse execution. The experimental results with four different benchmarks show a wide range of speedups from 1.3X for a small program with few data inputs to six orders of magnitude (1,928,500X) for 400x400 matrix multiply. Furthermore, our technique is very memory efficient. Our benchmark measurements show between 3.4X and 2240X memory overhead reduction as compared to our implementation of the same features using traditional approaches.

1. Introduction

Debugging is almost always an inevitable part of software development. Many programmers spend as much or more time on debugging a program as they spend on writing the program. Therefore, to increase software development efficiency, it is essential to cut down the time spent on debugging.

Debugging can be 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. One of the most time consuming parts of interactive debugging is trying to locate a bug. A major contributing factor to this time loss is extra re-executions programmers have to go through if they (by executing the program too far) lose the program context which leads to a bug.

In our previous work [3, 4], we introduced an assembly instruction level reverse execution methodology for debugging. By this methodology, a programmer can execute a program backwards at the assembly instruction level to any point in the program's execution history without having to restart the program. This potentially reduces the debugging time significantly.

Assembly level reverse execution of a program is realized by executing a reverse version of that program. This reverse version recovers the states the original program modifies. In short, our algorithm, which we name the *reverse code generation* (*RCG*) algorithm, generates a reverse version of a program by constructing the inverses of the assembly instructions in the program. Typically, the RCG algorithm constructs the inverse of an instruction such that the constructed inverse regenerates the state destroyed by that instruction without requiring any state saving. In occasional cases where a state regeneration is not possible, however, the RCG algorithm resorts to state saving. Therefore, the RCG algorithm not only provides reverse execution at the assembly level, but also saves considerable memory.

Reverse execution at the assembly instruction level enables an extremely fast backup when the target point at which the programmer wants to recover the state is close to the current program point. However, if the target point is far away from the current program point in execution, undoing each and every instruction inbetween might be tedious. In such a case, it might be useful to undo only those instructions that are related to the bug(s) in the program.

In this paper, we extend the RCG algorithm with a powerful debugging aid which is called *dynamic slicing* [2, 12]. Dynamic slicing isolates the statements that influence the value of a variable at a program point in a specific execution instance of the program. When the programmer realizes that a variable has an incorrect value at a certain program point, he or she can reverse execute the program along the corre-

sponding 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 rerunning the application with the dynamic slice specified.

The contribution of this paper is two-fold. Our technique not only provides a faster reverse execution via dynamic slicing, but also contains an advantage which is not offered by traditional dynamic slicing methods [2, 12]. The traditional dynamic slicing methods require runtime control flow information from the execution of a program to remove the redundant instructions that are irrelevant to a slice. However, the RCG algorithm can dynamically reconstruct this control flow information during reverse execution. This is achieved by the control flow predicates in the reverse program. The instructions irrelevant to a dynamic slice are simply skipped by these control flow predicates.

The outline of the paper is as follows. Section 2 describes the related work. Section 3 introduces the basics behind dynamic slicing and gives an overview of the RCG algorithm. Section 4 explains how we extend the RCG algorithm to provide assembly instruction level reverse execution along a dynamic slice. Section 5 presents the experimental results. Finally, Section 6 concludes the paper.

2. Related Work

Program slicing was first proposed by Weiser [19]. His approach statically determines the program statements that potentially affect a variable at a certain statement. Later, the concept of dynamic slicing was introduced by Korel and Laski [12]. 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.

However, none of these approaches provide a way to obtain the runtime values of variables in a program slice without at least re-executing the slice. Agrawal et al. added an execution backtracking approach to their debugger tool SPYDER which also supports dynamic program slicing [1]. This backtracking approach statically associates with each assignment statement a set of variables, called a change-set, which is modified by that statement. Then, during program execution, the associated variables in the change-set are recorded for rollback. However, as opposed to what we propose in this paper, the backtracking is not performed along a dynamic slice but along the whole program. Moreover, this approach may cause large memory and time overheads during program execution, especially with programs which modify the state frequently.

For the purpose of obtaining previously destroyed program states, there have been many other techniques proposed in literature [7, 8, 9, 15]. However, these techniques also lack a reverse execution capability along a dynamic program slice.

The most relevant work was carried out in the domain of functional languages [6]. Booth and Jones associate each variable in a program with a story tag which includes the history information about how the variable is computed. 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 large memory usage as the variable stories are built by pure state saving.

In terms of extraction of a dynamic slice, our method is similar to forward dynamic slicing algorithms [5, 13]. In a forward dynamic slicing technique, a dynamic slice is obtained during execution of the program, which is similar to our approach where a dynamic slice is extracted during reverse execution. Therefore, the runtime trace information is kept bounded as in the case of our technique. 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. On the contrary, our algorithm extracts a dynamic slice for the selected instruction only, which potentially results in faster calculation of a dynamic slice.

In dynamic slicing techniques that depend on an execution history, the processing of a complete program trace also constitutes a large memory usage. Zhang et al. indicate in their paper that this memory usage may go up to 9GB for 134.perl in the Specint95 suite. To reduce this 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 [21]. In this sense, our approach is orthogonal to the approach of Zhang et al. In our approach, the information required to build a dynamic slice is not extracted from an execution history, but is mainly reconstructed by our RCG algorithm.

3. Background

In order to understand the benefits of dynamic slicing for assembly level reverse execution, we need to first understand what program slicing performs and how our instruction-level reverse execution method works. The following two subsections give overviews of dynamic slicing and the RCG algorithm, respectively.

3.1. Program Slicing

There are two major types of program slices proposed in the literature. These are static program slices [10, 18, 19] and dynamic program slices [2, 12].

Definition 3.1 *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 3.2 *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, we can alter the definition of a dynamic slice to apply to an assembly level program as follows:

Definition 3.3 *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_j 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 [11, 18].

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.

3.2. Overview of the RCG algorithm

The RCG algorithm is an assembly level algorithm. It analyzes a program instruction by instruction and generates the reverses of the instructions to build a reverse program. For ease of analysis, multi-procedure programs are first chopped off into single-entry single-exit program regions, which we call *program partitions*, and each program partition is analyzed and reversed separately. Then, program partitions are connected using a state saving method [4]. Due to space limitation, this section explains reverse code generation within program partitions only. Interested reader can find details of reverse code generation across program partitions in [4]. Please note that in the rest of this paper, the word "instruction" refers to an assembly instruction.

We call the reverse of an instruction a *reverse instruction group* (*RIG*). We use the term "group" because a RIG may be composed of multiple assembly instructions. A RIG reverses the effect of an instruction by recovering the values that are overwritten by that instruction. For this purpose, we use data and control flow information to figure out where a value is defined, where it is used and how it reaches a particular instruction before being destroyed. Once we have that information, we can recover a value by using a combination of three methods: (i) the value can be recalculated during reverse execution by re-executing the original definition, which we call the *redefine technique*; (ii) the value can be extracted from a previous use during reverse execution, which we call the *extract-from-use technique*; and (iii) the value can be saved during forward execution and then restored during reverse execution, which we call the *state saving technique*.

Note that the recovery of a value may require the recovery of other values. For example, this would happen if the value of a register were to be regenerated using the value of another register which is also overwritten. Therefore, the redefine and extract-from-use techniques are usually applied recursively where the values in the dependency chain are recovered one after another.

Please also note that the current implementation of the RCG algorithm assumes sequential input programs running on a single processor only. Moreover, all operations requiring external input such as file I/O are reversed by the state saving technique explained above.

The following example illustrates how a reverse program is built from individual RIGs. For a detailed explanation of the RCG algorithm, we refer the reader to our previous work [3, 4].

Example 3.1 Figure 1 shows an example program and the corresponding reverse program. Each instruction (except control flow instructions and the instructions shown in bold) in the original program and the corresponding RIG in the reverse program are marked with the same index. Note that the instructions that are shown in bold are extra instructions

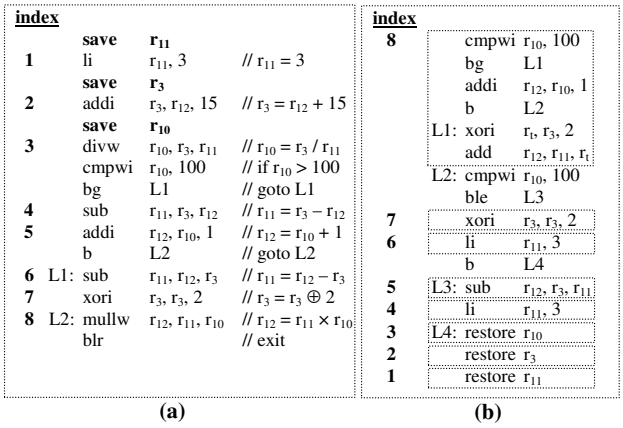


Figure 1. (a) A program in PowerPC assembly (b) The corresponding reverse program

that are inserted to the original program for state saving; thus, these instructions do not have associated RIGs. In addition, 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, we assign 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. Throughout this example, we use the notation i_x to refer to an instruction with index x and the notation RIG_x to refer to a RIG for the instruction with index x.

Let us have a close look at some generated RIGs. Consider RIG_7 for instance. RIG_7 can be seen in Figure 1(b) in the rectangle enclosing "xori r_3, r_3, 2" to the right of index 7. RIG_7 recovers the value of r_3 that is overwritten by i_7. For this purpose, RIG_7 employs the extract-from-use technique which extracts the overwritten value of r_3 from the use by i_7.

As another example, consider RIG_8. In RIG_8, we want to restore r_{12} to the value it held prior to execution of i_8 in Figure 1(a). There are two possible values that r_{12} might have carried before being overwritten by i_8. One of these values is defined at i_5 and the other value is defined prior to this section of code but is *used* in i_6. Each of these values reaches i_8 along a different path. Therefore, in RIG_8, we generate two sets of instructions where the first set (the third instruction in RIG_8) recovers r_{12} by re-executing i_5 and the other set (the last two instructions in RIG_8) recovers r_{12} by extracting its value from i_6. Then, we combine these sets using a conditional branch which determines along which path i_8 is dynamically reached. Note that the extraction of the value of r_{12} from i_6 is handled in two steps: we first recover the value of r_3 used in i_6 into a temporary register r_t and then use r_t to recover the value of r_{12}.

Finally, consider i_1. The value i_1 overwrites (the value of r_{11}) is input to the program (i.e., defined outside) and is not used anywhere before being overwritten. This implies that this value can be recovered neither by the redefine technique nor by the extract-from use technique. Thus, the RCG algorithm resorts to state saving which is performed by the inserted save instruction before i_1. Then, the generated RIG, RIG_1, simply restores the value from the saved record.

As seen in the figure, RIGs are placed in bottom-up order in the reverse program. In other words, the RIG for the first instruction goes to the very bottom of the reverse program, the RIG for the second instruction goes on top of the previous RIG and so on. This follows intuitively from the fact that an instruction that is executed first should be undone last. Another interesting note is that the RIGs are combined by conditional branch instructions which establish a control flow in the reverse program according to the control flow of the original program. For instance, i_8 can be reached in two ways: either from i_7 or from i_5 (via the branch "b L2") depending on the predicate calculated by "cmpwi r_{10}, 100". Therefore, after RIG_8, the RCG algorithm uses the same predicate to direct the control either to the reverse of i_7, RIG_7, or to the reverse of i_5, RIG_5. However, note that if r_{10} in "cmpwi r_{10}, 100" were also modified before i_8, then we would first have to recover r_{10} in the same way we recover any other register. □

4. Methodology

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 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 explain 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 through that slice. Consider the following example.

Example 4.1 Figure 2 shows a PowerPC assembly code piece with five instructions. The instructions that reside in the dynamic slice according to the slicing criterion C = (r_1 = 0, r_3, (addi r_3, r_3, 1)[1]) are enclosed in rectangles. Thus, 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

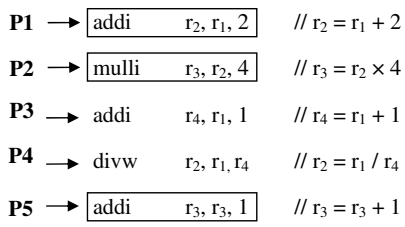


Figure 2. A code piece and a dynamic slice

'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 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.

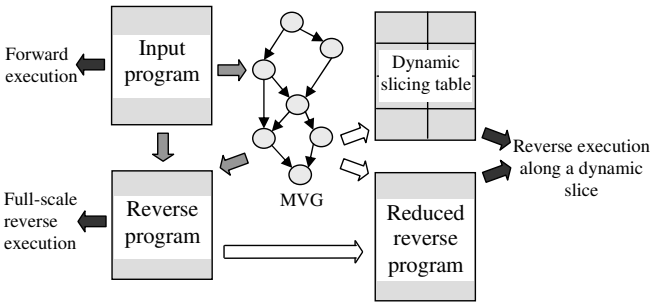
However, if we were to undo only the instructions within the dynamic slice to reverse execute through that slice, we would not have retrieved the value of r_2 at point **P1** because r_2 is overwritten by the fourth instruction which is outside the slice. Therefore, while trying to obtain the values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

As Example 4.1 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. The next section presents the extensions we make to the RCG algorithm following this instruction removal approach. Hereafter, we refer to the RCG algorithm with dynamic slicing support *RCG with Slicing* or *RCGS*.

4.1. The extensions to the RCG algorithm

The extension to the RCG algorithm for dynamic slicing support consists of two parts. The first part is a static analysis part. The second part is debugger support which incorporates dynamic information.

Figure 3 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 our previous work. Then, the programmer can start debugging the program with full-scale reverse execution support. When the programmer decides to obtain a dynamic slice, 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. However, once a dynamic slice is chosen, the programmer may have to wait for several seconds or longer (depending on the size of the input program) for the execution of static analysis required for the chosen dynamic slice. Nevertheless, this waiting time is still negligible compared to the overall time spent on debugging.



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 3. A diagram of the RCGS algorithm

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 3). 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.

On the other hand, the 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.

The RCGS algorithm performs the extended static analysis over a derivative of a *value graph* [17] which we call the *modified value graph* (*MVG*)–see Figure 3. The MVG

is a replacement for the *directed acyclic graph* (*DAG*) presented in our previous work [3]. An MVG is generated only once during the generation of the complete reverse program. Afterwards, the generated 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 MVG to statically find the definitions which might influence L at I. We call such definitions the *potentially influencing definitions*.

Ideally, the reduced reverse program should be composed of only the RIGs that recover the potentially influencing definitions. However, as explained in Section 3.2, 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 following subsections, we introduce the MVG and the details of its usage to extract a reduced reverse program.

4.2. The modified value graph (MVG)

The modified value graph is a directed graph $G = (N,E)$ where N is a set of nodes and E is the set of edges between these nodes. The details related to the construction of an MVG that are common to the construction of a DAG can be found in [3, 4] and thus will not be explained here.

Figures 4(a) and 4(b) show a sample program and its MVG, respectively. Each node in an MVG represents a definition in *static single assignment* (*SSA*) form [17]. The SSA form distinguishes each distinct definition of a register or a memory location by giving a different name to that definition. Figure 4(b) shows these names in the form of $(r|m)_x^y$ where r indicates a register, m indicates a memory location, x (x = 0, 1, 2, ...) is an index that distinguishes the physical location and y (y = 0, 1, 2, ...) indicates the distinct name given to a definition of that location. For instance, r_1^0 is the name given to the initial definition of register r_1. Also, in Figure 4(b), we mark a definition with the number of the instruction by which the definition is made.

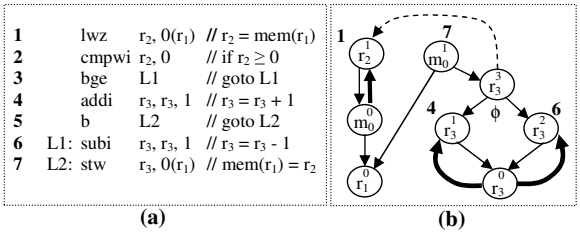


Figure 4. (a) An example program (b) The corresponding MVG

If more than one definition reaches a confluence point of edges in a program's control flow graph, the SSA form generates a pseudo definition which combines these reaching definitions at the confluence point. A pseudo definition generated at a confluence point selects one of the combined definitions depending on the predicate which determines on which path the confluence point is dynamically reached. In an MVG, we designate with μ the pseudo definitions that are generated at loop entrances and with ϕ the pseudo definitions that are generated at any other confluence point. For instance, in Figure 4(b), similar to SSA form, we say $r_3^3 = \phi(r_3^1, r_3^2)$ meaning that r_3^3 is a pseudo ϕ-definition which selects either r_3^1 or r_3^2 (depending on the predicate $r_2 \geq 0$). 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 static definitions in a program. A dynamic dependence graph which is used by Agrawal and Horgan for dynamic slicing [2], on the other hand, can be extremely large as it may contain different nodes for different instances of a statement.

The information required to find the essential definitions in a program is obtained from the edges of an MVG. There are three kinds of edges in an MVG:

1. Use-definition edges: These edges designate a data dependency between two values in the graph. Figure 4(b) shows these edges as solid thin lines. There is a directed use-definition edge $e_{ij} \in E$ from node $n_i \in N$ to node $n_j \in N$ if n_i and n_j are the values for target and source operands of an instruction α, respectively, or (2) n_i is a memory value and n_j is a register value determining the location of n_i, or (3) n_i is a pseudo definition node and n_j is one of the definitions being combined under that pseudo definition.

2. Pseudo definition-predicate variable edges: These edges are used for extracting the control dependency information between the nodes. As explained above, the μ nodes and the ϕ nodes select one of their combined definitions according to a predicate. A directed pseudo definition-predicate variable edge extends from a μ node or a ϕ node to the values in the controlling predicate expression. For example, the dotted line in Figure 4(b) is a pseudo definition-predicate variable edge which connects the pseudo definition r_3^3 to the value r_2^1 used in the controlling predicate expression for r_3^3.

3. Definition-recovering edges: These edges help us determine the definitions that are required for recovering the potentially influencing definitions. As the name indicates, a definition-recovering edge combines a definition δ_i to the definition δ_j that is used to recover δ_i. The solid thick lines in Figure 4(b) are examples of these kind of edges. For instance, since memory value

m_0^0 is equal to r_2^1, this memory value can be recovered by using r_2^1. Assuming that we choose to recover m_0^0 from r_2^1, we place a definition-recovering edge from m_0^0 to r_2^1.

4.3. Generating a reduced reverse program using an MVG

As mentioned in Section 4.1, we generate a reduced reverse program by using an MVG. This section explains the details behind this process.

Listing 1 shows the pseudo code for the reduced reverse code generation. 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-recover_edge(k,j) designates the j^{th} outgoing definition-recovering edge of node k. Follow_edge(n, e) returns the node connected to node n via edge e.

Given a dynamic slicing criterion C = (X, L, I^q), the RCGS algorithm first determines 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 1. Let us designate the found node with n. As mentioned in the previous subsection, 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 3.1, 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 to the set of essential definitions as defined in Section 4.1 (lines 5, 6, 10 and 11). On the other hand, when each such node m is added to the essential definitions set, the RCGS algorithm also finds the nodes that are connected to m via definition-recovering edges (lines 13 and 14). The found nodes correspond to the definitions that are required for recovering m. Therefore, these nodes, if not already added, are also added 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 RIGs needed to generate the reduced reverse program are determined, there are some optimizations we can carry out in the reduced reverse program. One optimization is that if the body of a loop becomes empty after the RIGs are removed from the reverse program, then the loop no longer remains necessary in the reverse program. Therefore, in this case, we remove the loop from the reverse code as well. The same is true for a conditional branch instruction whose branch targets become empty (line 23). Note that when RIGs are removed, some instructions may shift in the reverse program; therefore, in such a case, we also update the branch target addresses accordingly (line 24).

Listing 1 Generate a Reduced Reverse Program

```
Inputs: A complete reverse program T and its MVG
        A dynamic slicing criterion C =(X, L, I^q)
Output: A reduced reverse program
begin
 1  Find node n which corresponds to definition of L at I
 2  worklist = n
 3  repeat
 4     n = worklist
 5     for i = 1 to | outgoing_dep_edges(n) | do
 6        k = follow_edge(n, outgoing_dep_edge(n,i))
 7        if k ∉ worklist then
 8           worklist += k
 9        end if
10        if k ∉ essential_defs then
11           essential_defs += k
12        end if
13        for j = 1 to | outgoing_def-recover_edges(k) | do
14           m = follow_edge(k, outgoing_def-recover_edge(k,j))
15           if m ∉ essential_defs then
16              essential_defs += m
17           end if
18        end for
19        worklist -= n
20     end for
21  until worklist = φ
22  Pick from T the RIGs which recover essential_defs
23  Remove loops with empty bodies and branches with empty targets
24  Update the target addresses of the remaining branches if necessary
end
```

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

Example 4.2 Figures 5(a) and 5(b) show an example program and the corresponding MVG, respectively. For clarity, we do not show definition-recovering edges in Figure 5(b). Instead, definition-recovering relationships between the nodes are shown by the table in Figure 5(e). As in Example 3.1, 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 5(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 5(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 5(c) and 5(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 annotate each destroyed definition in Figure 5(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 5(b), we annotate the node of r_5^0 with "RIG_9" (note that some nodes in Figure 5(b) do not have associated RIGs because these nodes either are not destroyed at all or do not

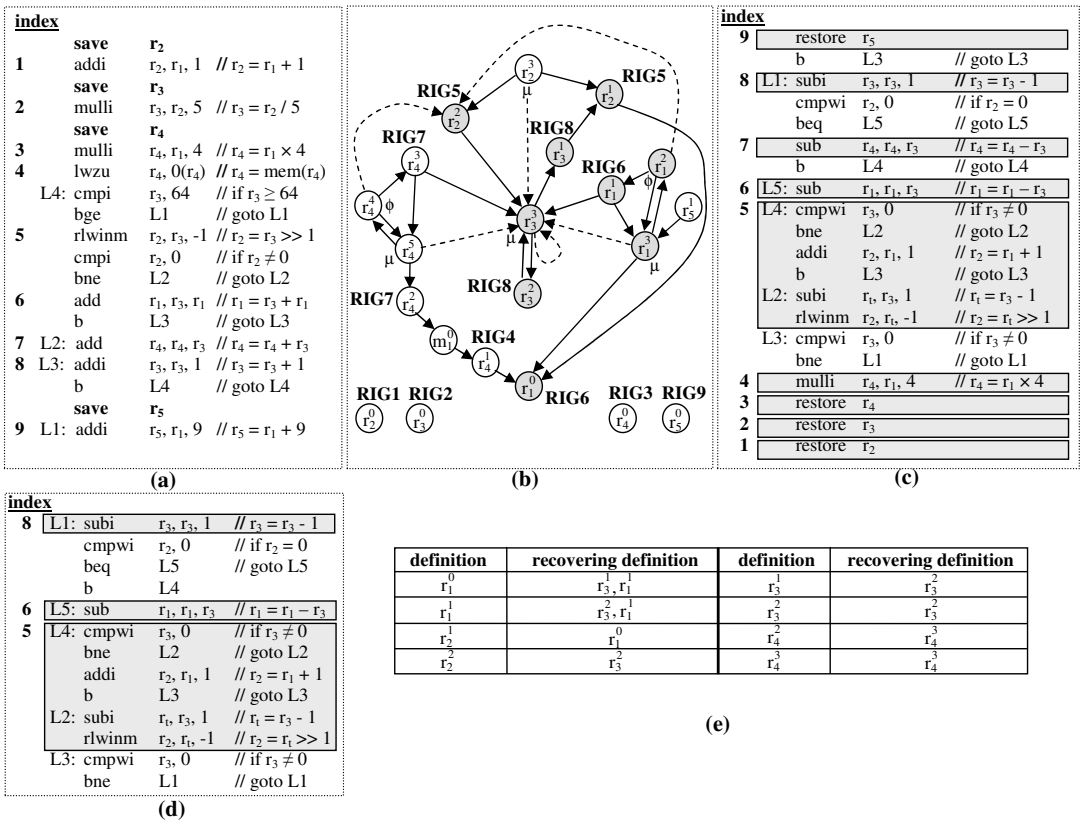


Figure 5. (a) An example program (b) The corresponding MVG (c) The complete reverse program (d) The reduced reverse program (e) Table showing definition-recovering relationships

represent real definitions but only pseudo definitions). As seen from Figures 5(c) and 5(d), the reduced reverse program includes only those RIGs that correspond to the essential definitions. □

5. Experimental Results

This section presents the measurements we performed. For experimentation, we used an evaluation board with a PowerPC (MPC860) processor and 4MB of memory. We used this board because the base RCG algorithm was implemented for the PowerPC target. The board is connected to a PC via a *Background Debug Mode* (*BDM*) interface [16]. The PC which runs Windows 2000 also hosts our debugger tool [3, 4] supporting instruction-level forward and reverse execution via a graphical user interface. The PC has dual 600Mhz Pentium III processors and 1GB of memory.

The benchmarks we used are matrix multiply, selection sort, an adaptive differential pulse code modulation (ADPCM) encoder from Media Bench [14] and a data compression utility (Compress). We could not use much larger benchmarks such as those in SPEC suite mainly due to memory limitation of the PowerPC board. However, this should not imply any scalability limitations for the RCG algorithm. As mentioned in Section 3.2, the RCG algorithm partitions the subject program and operates on each partition separately. Thus, rather than the number of partitions in a program, instead the characteristics of the partitions (e.g., the percentage of the instructions that require state saving for being reversed) play an important role in determining the efficiency of the RCG algorithm. Nevertheless, even though the static instruction count of our benchmarks is small (up to 4000), the number of dynamically executed instructions is quite large. This number ranges between 500 instructions (for selection sort with 10 inputs) and 896 million instructions (for 400x400 matrix multiply).

We first compared full-scale reverse execution against reverse execution over a dynamic slice in terms of overall execution time. In this experiment, the matrix multiply was performed over two 4x4 matrices, the selection sort was performed over an array of 10 integers and the ADPCM encoder and Compress were run over 128KB input data. We experimented over three different dynamic slices for each benchmark. For matrix multiply and ADPCM encoder, we took the slices for two different registers, while for selection sort and Compress, each slice was taken for a single register. The execution time measurements were performed

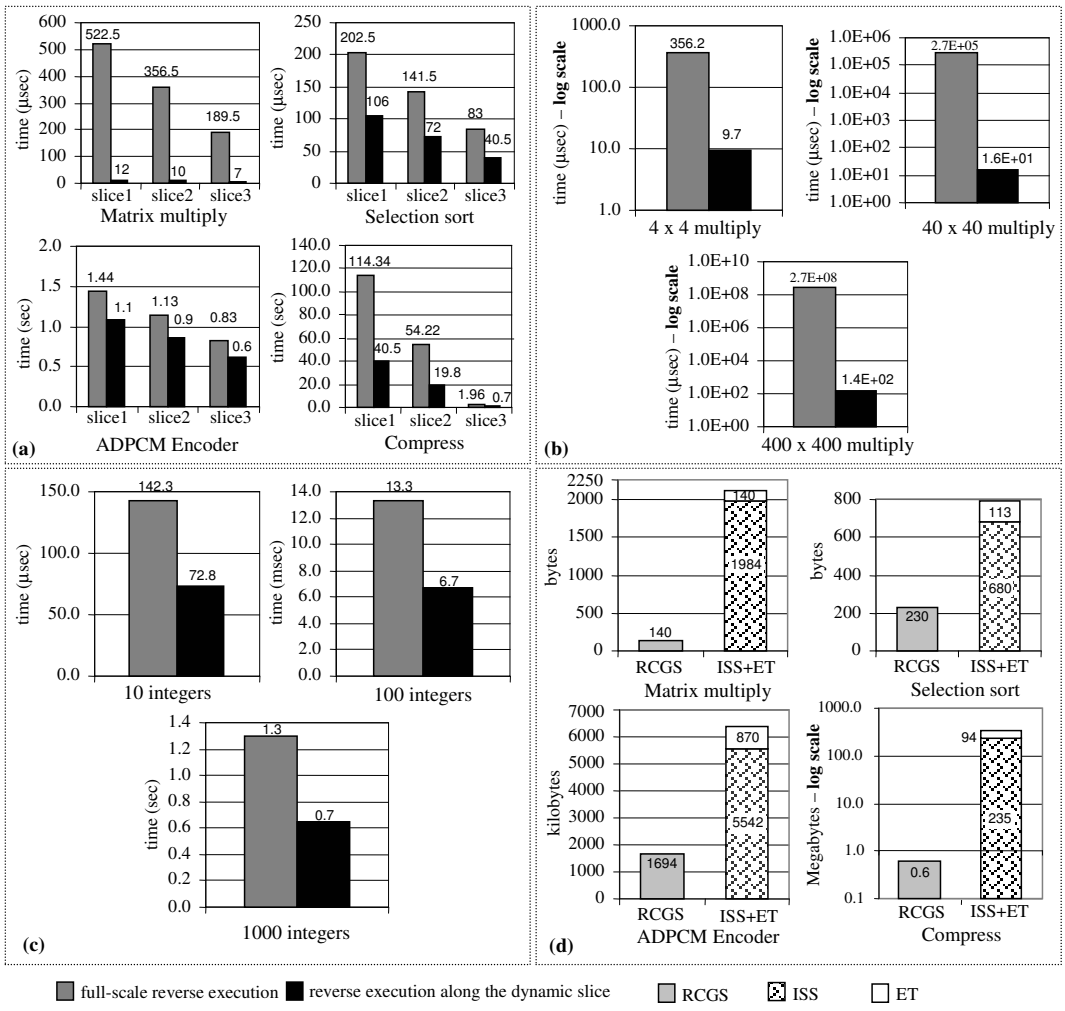


Figure 6. (a) Reverse execution time comparison (b) Reverse execution time comparison of matrix multiply with different matrix sizes (c) Reverse execution time comparison of selection sort with different input array sizes (d) Runtime memory requirement comparison

via the decrementer counter of the PowerPC processor in a nonintrusive way. Figure 6(a) shows the results for the four benchmark programs. The average speedups achieved for matrix multiply, selection sort, ADPCM encoder and Compress 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 reverses of the other benchmarks. The average code size reduction from the complete reverse program to the reduced reverse program is the biggest for matrix multiply with a factor of 6.6, while the code size reduction factors for selection sort, ADPCM encoder and Compress are 3.1, 1.4 and 1.5, 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 6(b) and 6(c). For instance, with 400x400 matrix multiply, the full-scale reverse execution takes around 4.5 minutes in average, while the reverse execution along a dynamic slice takes only an average of 141 microseconds.

The RCGS algorithm provides important debugging support by implementing reverse execution along a dynamic slice with little runtime trace information. To show this, we compared the RCGS algorithm with one of the most common reverse execution techniques, the incremental state saving technique (ISS) [20] in terms of the average run-

time memory requirement. Moreover, we measured the average runtime memory requirement of an execution trajectory (ET) which is usually used to obtain dynamic slices in traditional way [2, 12, 21] (Please note that, as observed by [21], forward dynamic slicing [5, 13], does not contain any explained results for us to compare against our results). Figure 6(d) shows the results from the benchmarks. In this experiment, the input sizes of the benchmarks were chosen to be the same as the sizes in our first experiment. The results indicate that compared to ISS+ET, the RCGS algorithm achieves approximately 15.2X, 3.4X, 3.8X and 548X reduction in memory overheads for matrix multiply, selection sort, ADPCM encoder and Compress, respectively. Due to loop effects, 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. Note that due to the limited memory on the PowerPC board, we performed these last set of measurements using circular buffers.

The static analysis of RCGS takes less than five seconds on the host PC for any of the benchmarks. Considering the total time spent on debugging, this time is negligible. On the other hand, the runtime computation overhead of slicing which mainly consists of the determination of actual control flow paths is included within the timing results presented.

6. Conclusion

In this paper, we have introduced a new approach to reverse execution along a dynamic slice. Specifically, our approach skips recovering the program state not required for reverse execution along a dynamic slice. Therefore, our approach provides a fast reverse execution at the assembly instruction level. Our technique does not require prior knowledge of the dynamic slicing criterion. Therefore, the programmer can start reverse executing a program along the desired dynamic slice without having to restart the program. Moreover, our 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 our technique very memory efficient.

References

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

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

[3] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. In *Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering 2002 (PASTE'02)*, pages 18–25, November 2002.

[4] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. Technical Report GIT-CC-02-49, Georgia Institute of Technology, Atlanta, GA, USA, 2002. http://codesign.ece.gatech.edu/publications/index.htm.

[5] A. Beszedes, T. Gergely, Z. M. Szabo, J. Csirik, and T. Gyimothy. Dynamic slicing method for maintenance of large C programs. In *5th European Conference on Software Maintenance and Reengineering*, pages 105–113, 2001.

[6] S. P. Booth and S. B. Jones. Walk backwards to happiness - debugging by time travel. In *Automated and Algorithmic Debugging*, pages 171–183, 1997.

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

[8] J. J. Cook. Reverse execution of java bytecode. *The Computer Journal*, 45(6):608–619, May 2002.

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

[10] R. Gupta and M. Soffa. A framework for generalized slicing. Technical report TR-92-07, University of Pittsburgh, 1992.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[12] B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[13] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *International Symposium on Software Testing and Analysis (ISSTA)*, 1994.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

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

[16] Motorola Inc. *MPC860 PowerQUICC Users Manual*, 1998.

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

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

[19] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

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

[21] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *IEEE/ACM International Conference on Software Engineering*, pages 319–329, 2003.