Execution Backtracking Using Reverse Execution of Machine Code

by

Paria Noorkami

B.Sc., Shahid Beheshti University of Tehran, Iran, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Applied Science in THE FACULTY OF GRADUATE STUDIES (Department of Electrical and Computer Engineering)

We accept this thesis as conforming to the required standard

The University of British Columbia

September 1999

© Paria Noorkami, 1999
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical & Computer Eng.

The University of British Columbia
Vancouver, Canada

Date Sep., 15, 1999
Abstract

Execution backtracking is the process of restoring the state of a program to an arbitrary point earlier in its execution history. It is used to facilitate program debugging. In this thesis, a novel execution backtracking approach is developed and implemented to assist the task of debugging software. The approach is demonstrated for structured C programs and exploits backtracking the program at machine code level. The approach has a lower run-time overhead than the existing approaches.

The execution backtracking approach is integrated with a diagnosis test bed that consists of a symbolic debugger, a C cross-compiler, a virtual machine and a symbolic reasoner. A symbolic debugger is used to debug the target executable which is instrumented with a C cross-compiler. Relative advantages of the approach is that it is language independent, it is able to backtrack in presence of pointer operations, and it supports true execution replay and dynamic slicing. A relative disadvantage of the approach is that it is only able to partially restore a program's data state.
Contents

Abstract ii

Contents iii

List of Figures vi

Acknowledgements viii

Dedication ix

1 Introduction 1

1.1 Program Debugging ........................................... 1

1.2 Execution Backtracking ..................................... 3

1.2.1 Advantages of Execution Backtracking .................. 4

1.3 Motivation ..................................................... 5

1.4 Contributions .................................................. 7

1.5 Outline of the Thesis ........................................... 8

2 Background 10

2.1 How Do Conventional Debuggers Work? ..................... 10

2.2 Execution Backtracking ..................................... 12

2.2.1 Notations and Concepts ................................. 12
List of Figures

1.1 Overview diagram of the diagnosis test bed ................................. 7

3.1 One-process configuration of running CDB ................................. 28
3.2 Two-process configuration of running CDB ................................. 28
3.3 Revised CDB configuration .................................................. 34

4.1 Example program 1 ............................................................. 41
4.2 Static control dependence graph of example program 1 .................. 42
4.3 Dynamic control dependence graph of example program 1 when input
    j is 2 ................................................................. 43
4.4 Example program 2 - instrumented code .................................... 46
4.5 Example program 2 - run-time data structures ............................ 47
4.6 Example program 3 - instrumented code .................................... 49
4.7 Example program 3 - run-time data structures ............................ 50
4.8 Example program 4 - instrumented code .................................... 52
4.9 Example program 4 - run-time data structures ............................ 53
4.10 Example program 5 - locations of _control_table entries .............. 57
4.11 Example program 5 - _control_table ...................................... 58
4.12 RCDG pseudo code for recovering if control statements .............. 59
4.13 RCDG pseudo code for recovering switch control statements ......... 60
4.14 Example program 5 - run-time data structures when i is 4 . . . . . . 61
4.15 RCDG pseudo code for recovering *while* control statements . . . . 62
4.16 Example program 5 - *while* run-time data structure . . . . . . . . 63
Acknowledgements

I am writing to express my gratitude for all the valuable support I have received while completing my masters degree in the Department of Electrical and Computer Engineering at the University of British Columbia.

Particularly, I wish to thank my supervisor, Dr. Greg W. Bond, who generously provided his time and expertise to me and my work. It has been such a pleasure to work with him and I am grateful for his commitment.

Equally, I am grateful to my parents, Pari and Morteza, and my sister, Maneli, who provided unconditional love and support at great distance.

Additionally, Ismail Lazoglu, provided very valuable comments and suggestions to me as I worked through my thesis and his participation was also appreciated.

As I come to the end of my time in the department, I shall miss the rewarding working relationships and friendships that have developed. In this regard, I say thank you to Buke Otkunc and Andrew Watkins who have become dear colleagues and friends.

My regards to my peers and my thanks to the department.

Paria Noorkami

The University of British Columbia
September 1999
To my parents, Pari and Morteza,

and

to my sister, Maneli.
Chapter 1

Introduction

This chapter introduces the thesis research topic. It reviews the conventional task of debugging programs and explains execution backtracking, which is a technique to assist with diagnosing software. The motivation and contributions of the research are also discussed. Finally, an outline of the thesis is provided.

1.1 Program Debugging

Debugging is the process of locating and correcting a program’s faulty behavior which has been revealed from the testing phase or its operation in the field.

Reading the code and program documentation, run-time monitoring and printing variable values are common techniques for debugging. However, these techniques are time-consuming and require much human effort. These problems are exacerbated when the program is poorly documented, not well commented, and when the program code is large.

Traditionally, symbolic debuggers are used only to gather information about the program’s actual structure and behavior; the diagnostic reasoning tasks are performed
manually by the programmer. In common debuggers, first, a user sets a breakpoint at a suspicious point of the program code and stops its execution at that point. Then, the programmer examines the program state (values of variables) at that point to check whether the observed behavior of the program matches its expected behavior. If any unexpected behavior is observed at that point, the programmer starts to think backward and repeats the process of testing program states at earlier points during the program's execution. By setting the breakpoints and checking the program states, the user reduces the search space of potential faults. The major drawback of this technique is that the reasoning tasks about the program's behavior and semantics are manually performed by the user. The debugger only provides the breakpoint facility and allows the user to check the values of variables at the breakpoints. Furthermore, by setting each breakpoint, the programmer needs to re-execute the program from its beginning. Therefore, the run-time overhead will grow considerably large; especially when the program is lengthy and complex. Program re-execution is simple when the program lifetime is short, program inputs are few, and the program is not interacting with its environment very much. On the other hand, re-execution is much more difficult for long-running programs with many inputs and for programs with non-deterministic execution environments (i.e. network I/O, file I/O, memory alloc/dealloc).

Attempts have been made to build tools to provide automated debugging assistance and to develop more efficient and affordable debugging techniques. One of these techniques is execution backtracking. Execution backtracking is briefly introduced in the following section.
1.2 Execution Backtracking

Execution backtracking is the process of restoring the state of a program to an arbitrary point earlier in its execution. It is a process that facilitates software debugging and makes it efficient and affordable. Execution backtracking allows the user to avoid re-executing the program for each new breakpoint. It also supports the natural diagnostic task of backward reasoning about program behavior.

Execution backtracking also supports execution replay and dynamic slicing. Execution replay [19, 18, 15] is the process of replaying the execution of a program exactly as it had executed in the first place. Usually, it is required to execute the program over and over again to track down the bugs. Instead, a user would like to re-execute a program from an arbitrary execution point, exactly as it had executed in the first place.

Re-execution of large programs that interact with the environment via system calls does not necessarily produce the same program states that occurred during the original execution. Therefore, an execution backtracking approach which does not require the re-execution of a program is guaranteed to restore the same program states that occurred during the original execution.

Dynamic slicing can be combined with execution backtracking to build a more effective debugging tool. Dynamic slicing [23] is a technique used in debugging which generates a subset of a program execution trace containing all the statements affecting the value of a variable in a specific trace of the program. It is useful for reducing the search space of potential bugs.

An execution backtracking facility in association with breakpoints and traces can be used in conventional debuggers to assist with the task of debugging. The programmer can set breakpoints at earlier points of program execution, and examine the earlier values of program variables at those points. Some earlier systems that
have provided execution backtracking [4, 21, 22] required an unbounded amount of storage. For example, an interactive debugging tool, Exdams [4], provided an execution replay facility by recording the complete execution history of the program on tape. The program can be re-executed through replaying the tape. Maintaining a complete execution history of program operations and their side effects requires a large, unbounded amount of space. In order to overcome the space problem, some systems use a bounded history list. However, this approach restricts backtracking to only the parts of the program whose execution history has been recorded.

Three more recent approaches to execution backtracking have been introduced by Agrawal et. al [2] in 1991 and Shapiro [19] in 1997. These approaches address the drawbacks of space inefficiency and limited backtracking. However, the approaches still require a considerable amount of information to be stored.

In Agrawal et. al’s first approach [2], the complete execution history and program states are stored. Although it is possible to store only a partial execution history, the trade-off is that limited backtracking will be provided. Overall, the first approach is not efficient because of high run-time overhead due to storing large amounts of information. Agrawal et. al’s second approach reduces the space problem by saving only program state for each logical block of the program. Although space cost is reduced to some extent, it puts constraints on backtracking.

1.2.1 Advantages of Execution Backtracking

Using conventional debuggers, when an error is manifested in the program, the programmer starts backtracking through the source code, mentally, one statement at a time, to analyze the effect of each statement and locate the origin of the error. In order to perform this task, the programmer has to determine the regions more likely to contain the fault, then set breakpoints and suspend execution at those points. Each
time a new breakpoint is set at an earlier location in the source code, the program must be executed from the starting point to that breakpoint. With an execution backtracking facility, one does not need to re-execute the program from the beginning to see the program state for each breakpoint. The programmer can step back one statement at a time until the fault is identified and localized.

To emphasize the importance of execution backtracking and how it eases the task of debugging, its merits are listed below:

- Execution backtracking reduces the time spent for debugging by allowing the user to restore the state of the program in reverse direction without running it again from the start of the source code.

- If the debugger also allows the user to modify the program state, then the execution backtracking facility could be used to do some what-if analysis over sections of the program.

- Different execution paths taken by the program could be easily examined using backtracking and state changing facility.

- Programmer can step back in the code without re-executing it, if programmer steps over a statement by not realizing its importance until some statements later in the execution.

- It provides a more automated debugging tool for the programmer to reason about the program.

1.3 Motivation

In the previous section, various backtracking approaches have been introduced. It can be observed that little research has been done in the area of execution backtracking.
The most recent work is by Agrawal et al [2] and Shapiro [19]. Although they have made some achievements in execution backtracking, there are still overall deficiencies that make their approaches inefficient and inadequate.

A benefit of the execution backtracking approach developed in this thesis is that it involves comparatively less run-time overhead than the existing approaches. Unlike Agrawal et al’s and Shapiro’s approaches that are required to store program states, no program states are recorded. Instead, only a small amount of control flow information is saved for each structured control statement.

The current approaches suffer from the problem of backtracking in presence of pointers. There have been attempts to overcome this problem, but in practice, the solutions have not been practical. The approach presented in this thesis exploits reverse execution of the program at the machine code level, therefore backtracking is possible in presence of complex pointer operations and composite data types.

An advantage of our approach is that partial program states can be restored to an arbitrary point of program execution without saving them during the original execution. This is especially useful for supporting execution replay.

In Agrawal et al’s and Shapiro’s approaches, instrumentation and run-time overhead are too large for use in the field. They are only practical in a program development environment. In contrast, our approach has much lower instrumentation and run-time overhead.

Existing approaches are highly language dependent since they must distinguish the scope of the variables and when their values are changed. This is especially complicated in the presence of pointers. The approach presented in this thesis is only language dependent to the extent of recognizing the program control flow structures. In general, the semantics of control flow constructs are simpler than the semantics of data flow and variable scopes, therefore our approach is relatively language independent because data flow analysis is not performed.
1.4 Contributions

In this thesis, a new approach to execution backtracking is developed and implemented. The approach is integrated with an existing symbolic debugger and a C language cross-compiler, and it is applied to structured C programs.

In order to examine the execution backtracking approach developed in this thesis, a program diagnosis test bed has been constructed which is an integration of different software components. Figure 1.1 shows the overview diagram of this diagnosis test bed.

The code generator of LCC [9], a C cross compiler, is modified to instrument the program executable. The instrumentation records the necessary control flow information of the program at run-time. The instrumented executable runs in a MIPS R2000 virtual machine [5] which is linked with CDB [11], a machine independent symbolic debugger, interfaced to the target. Modifications have been made to the MIPS simulator to link it with a part of the debugger. These modifications provide
operating system independency for CDB, and allow the two different parts of CDB to run on machines with different endianness. The CDB debugger, which provides the user interface, passes its queries to the target and receives responses from the target via interprocess communication. A MIPS R2000 symbolic reasoner [20, 12] is used to execute the machine code of program in reverse. It uses the final memory and registers contents of the MIPS virtual machine, and the recorded control flow information to restore partial program states to an arbitrary point during program execution.

The execution backtracking approach supported by the platform exploits reverse execution of the program at machine code level therefore, it allows backtracking in presence of complex pointer operations. Using the symbolic reasoner, it is possible to partially restore the data state of the program to an arbitrary point of its execution without saving program states during original execution. Only a small amount of control flow information needs to be saved, and since the semantics of program control flow constructs are simple, the approach is highly language independent.

1.5 Outline of the Thesis

Following the introduction of the thesis topic in the first chapter, Chapter 2 explains how conventional debuggers work and why they are not efficient. It describes the concept and advantages of using execution backtracking for debugging software. In addition, it reviews the related work on using execution backtracking for debugging software and discusses the drawbacks of the existing approaches. Finally, the enhancements that are achieved in the execution backtracking approach of this thesis are stated.

Chapter 3 introduces the software components that have been used to build the debugging test bed and justifies why each software component has been chosen. It
provides an overview of CDB, the symbolic debugger, that has been used in the thesis. In addition, it discusses the modifications that have been made to CDB for the purpose of this research.

Chapter 4 describes the execution backtracking approach developed in this thesis. It briefly introduces the concept of control flow dependence relations of programs. Moreover, it explains the design of the different development phases of the execution backtracking approach in detail. Finally, the advantages and drawbacks of the approach are discussed.

The results of the research presented in this thesis and recommended future work are summarized in Chapter 5.
Chapter 2

Background

In this chapter, we describe how conventional debuggers work, and provide an overview of work that has been done in the area of software debugging using execution backtracking. We explain why common debugging tools are not efficient, and how executing backwards can assist diagnosing software. Current symbolic debuggers only provide information about the program's actual structure and behavior, they do not help the programmer to reason about the program to identify the source of errors. Although, execution backtracking is a novel and important area of research for building automated debugging tools, there has been little research in the area of execution backtracking. In this chapter, two published approaches by Agrawal et al. [3, 2] are discussed in detail along with their merits and flaws. In addition, it is pointed out why the approach presented in the thesis has advantages over the existing ones.

2.1 How Do Conventional Debuggers Work?

Debugging is one of the indispensable and costly phases of software development life cycle. Several tools, from hexadecimal dumps of program state, to window and
mouse-based interactive debuggers, are available to simplify fault localization.

Common debuggers provide breakpoint and trace facilities for debugging software. Using these facilities, the natural process of debugging can be outlined as follows:

1. Stopping the execution of the program at a point of interest
2. Getting a snapshot of program state (values of variables) at that point
3. Comparing the observed state with the expected one
4. If an unexpected behavior is observed, start thinking backwards, deducing the conditions under which the program produced incorrect values for variables
5. Starting the process again by re-executing the program from the beginning until the source of error is discovered

Note that, only the first two phases of this process are done with the assistance of the debugger, the rest are manually performed by the programmer. The information used by the programmer consists of the source code, the program's inputs and the erroneous output for the specific execution trace of the program. After observing the erroneous output, the programmer starts thinking backwards to narrow down the search space of the source of error. By setting breakpoints at suspicious points in the program and rerunning it from the beginning, the process is repeated until the bug is localized and identified.

There are certain deficiencies of this natural and traditional debugging approach:

- It makes the debugger a semi-automated tool; most of the tasks are done, manually, by the programmer.
- Since the process of debugging is not automated, the person performing the task of debugging is required to have some knowledge about the program.
- It is cumbersome and complex for lengthy and complicated programs.

- It has high run-time overhead since the program has to be re-executed from the beginning for each breakpoint.

Naturally, these deficiencies make the debugging procedure inefficient and inadequate. Therefore, a more efficient and affordable approach is required. Execution backtracking in association with breakpoints and traces is described in [3, 2] as a new technique for debugging software programs. In [3, 2], slicing and backtracking are combined in a tool called, Spyder, to enhance the process of debugging. Program slicing, first introduced by Weiser [24], is a technique to reduce the programmer’s search space for finding bugs. A slice on a variable includes all the statements affecting the value of that variable. Backtracking is a technique for performing the natural reasoning task of identifying the bug. In the thesis only backtracking is related to our work and is reviewed. Program slicing is left for the future work. Program slicing can be combined with execution backtracking to build a more efficient debugging tool.

### 2.2 Execution Backtracking

The fundamental idea behind execution backtracking is running the program in the reverse direction. Two of the more advanced and efficient approaches by Agrawal et al. [2], are discussed and evaluated in this section.

#### 2.2.1 Notations and Concepts

Concepts that are used to describe the execution backtracking approaches are as follows:

**Program state:** The state of the program at a particular point of execution consists
of two things, values of variables in the program at that time, and the location of the program control.

**Program block:** A program block or a composite statement is a logical atomic block of a program and constitutes one or more program statements. For example, the body of a `while`, or an `if` construct is a program block.

**Change-set:** Change-sets are defined for both simple and composite statements.

- The change-set of a simple assignment statement consists of the original values of all variables whose values are modified by that statement. In other words, it includes the original values of the variables that appear on the left hand side of that assignment statement.

- The change-set of a composite statement (program block) is the union of change-sets of its constituent assignment statements. It includes the original values of all variables that might be modified by that program block. If the function computing the change-set is denoted by $C$, source statements are symbolized by $S$, $S_1$ and $S_2$, and a boolean expression by $cond$, the change-sets of some composite statements can be computed as follows:

$$C(S_1; S_2) = C(S_1) \cup C(S_2)$$

$$C(if \ cond \ then \ S) = C(S)$$

$$C(if \ cond \ then \ S_1 \ else \ S_2) = C(S_1) \cup C(S_2)$$ \hspace{1cm} (2.1)

$$C(while \ cond \ do \ S) = C(S)$$

Executing a program statement causes one program state to be transformed into another. The type of the transformation depends on the statement. Due to state transformation, the control location and value of one or more variables are modified.
For example, an assignment statement modifies its change-set because the value of the variable on its left hand side is changed. It also modifies the program control to be the successor statement. Conditional (if – then – else) and loop (while – do) predicates only modify the program control. A readI/O statement only modifies its change-set (value of the variable to be read) and writeI/O does not affect the program state at all. All of these conclusions are based on the assumption that the program state excludes the state of the program’s operating environment.

In order to reverse the execution, the earlier states of program need to be restored. Restoring the state of a program without re-executing it, requires storing program control and variable change-sets at each statement.

Two different approaches, namely the Execution History Approach and Structured Backtracking, are introduced in [2]. In the following subsections, these approaches explained by Agrawal et al. [2] are reviewed.

2.2.2 Execution History Approach

In this method introduced by Agrawal et al. [2], the state of program is stored as follows:

1. The execution history of program control locations, which is the sequence of program statements that are executed, is recorded.

2. For each program statement, its change-set is stored.

By traversing this information in the reverse direction, the earlier program states can be restored, therefore, backward execution can be performed. The programmer can backtrack to any statement saved in the execution history and retrieve the values of change-set variables at that statement. The programmer can continue backtracking to localize and identify the source of error.
Although this approach gives considerable power to the programmer in the task of debugging, it has overall deficiencies that makes it inefficient and expensive. Some of these flaws are as follows:

- The storage needed to record the program states is not bounded at compile-time. Since run-time input can determine the number of iterations of a loop construct, the execution history length is generally not bounded at compile-time. For each statement in a `while` loop, its change-set has to be stored for the number of iterations of that `while` construct. Thus, the execution history of such a construct can grow very long. To reduce the run-time space cost of this approach, it is possible to save only a partial record of program control flow information. This makes the approach more affordable and useful for programs with long lifetimes. Also, putting an upper bound on the number of iterations of `while` constructs can bound the run-time storage. The trade-off of these enhancements is that they provide limited backtracking.

- A potentially enormous amount of information (program state per statement) needs to be stored during program execution. Therefore, it can be said that this approach is quite time and space inefficient.

- The implementation of this approach is completely supported by the debugger. In order to perform execution backtracking, the programmer must execute the program first time under the control of the debugger. It is feasible to instrument the program code with assistance of a compiler to record the program states. Although, because a lot of information has to be recorded in different points of the program during execution, run-time overhead in both space and time will be huge.

- This approach is highly language dependent. It is necessary to determine when
values of variables are updated and which change-sets need to be modified. When the program structure is complex and pointers are widely used in the program code, determining the variables scope and their update time can be very costly and complicated.

- Finding the change-set can be costly and complex; especially when pointers and composite data types are widely used in the program code. The simple C program fragment below reveals the problem:

```c
p=&a[0];
*(p++)=2;
```

In the second assignment statement, one would think the change-set only includes pointer p, but the change-set also includes a[1], the second element of the array. This is because in the second statement, incrementing pointer p makes it point to a[1], the second element of array a. Therefore, it can be easily seen that finding the accurate change-set requires considerable effort which can be very costly and complex.

In [1], Agrawal et al. have introduced a solution to solve the problem of pointers and composite data types. In their solution, they associate a tuple (address, length) with each variable. The pointer and composite data type evaluation is performed by checking if the tuple of a variable overlaps (partially or completely) with another one. Although, in theory, this approach can handle the problem of pointers and composite data types, in practice, it has some drawbacks that makes it costly and inefficient. Some of these drawbacks are listed as follows:

- This method has a large run-time overhead. For pointers, composite data types and variables that require dynamic memory allocation, calculating
the tuples is performed at run-time which largely increases the program execution time.

- This method is highly language dependent. Calculating the tuples requires a mapping from variables' high level representation to their low level tuple representation.

- When pointer arithmetic operations and variable dynamic memory allocations are performed in the program code, finding the tuples can be very complex and costly.

- It has considerable storage cost. Variables' tuples need to be stored for all the variables.

2.2.2.1 Incremental Replay Debugging

Shapiro [19] has built on an approach similar to Agrawal et al.'s Execution History approach. It improves upon Agrawal et al.'s approach by supporting backtracking in the presence of asynchronous interrupts, and interactions with operating system such as file I/O.

2.2.3 Structured Backtracking

Agrawal et al. in [2] have introduced a revised and enhanced approach, called Structured Backtracking, to overcome the drawbacks of the Execution History Approach. In the new approach, program state is stored as follows:

1. No control flow information is saved.

2. A change-set is stored for each program block, not each statement.

As described in Section 2.2.1, the change-set of a composite statement like \( if \) or \( while \) includes the original values of all variables whose values could be modified...
during execution of that statement. For example, the change-set of a *while* loop consists of all variables that could be modified if the loop body is executed one or more times. Only one copy of the change-set is stored for loop constructs, no matter how many times they are executed. This copy is overwritten in each new iteration and contains the variables' recent values.

The new approach has advantages over Execution History Approach. Not only is the amount of stored information reduced in the new approach, but also the storage required is bounded at compile-time. It is claimed that the run-time storage is bounded at compile-time because the number and size of change-sets associated with program blocks are known at compile-time. However, the size of a change-set that includes the values of the variables which require dynamic memory allocation might not be bounded at compile-time.

Despite the advantages of Structured Backtracking, Agrawal et al.'s new approach [2] has the following flaws, and is therefore still inefficient;

- It puts constraints on backtracking, which restricts the programmer in performing backward execution;

1. One can not backtrack from a statement outside of a composite statement to inside of it. This restriction is due to the lack of control flow information. There is no information recorded to indicate which program blocks have been executed. For backtracking from outside of a program block to inside of it, the programmer has to first backtrack to the beginning of the program block and then execute forward to inside of that block.

2. One can not execute backward from a statement inside of a loop body to a statement in an earlier iteration of it, but can do so for the same iteration. This limitation exists because only one change-set is stored for each loop construct.
3. One can not backtrack from outside of a procedure call to inside of it. This is because the body of a procedure call is considered as a program block. The same reason stated in the first drawback above holds for backtracking into procedure calls.

- Since the Structured Backtracking Approach may require forward execution, it is not possible to use it to support execution replay. Execution replay is the process of replaying the execution of the program exactly as it had executed the first time. For programs that execute system calls, it is not guaranteed that the system call behave the same way each time it is executed. For example, each time a file read operation is performed, its file descriptor is incremented.

- Although there are fewer change-sets stored during program execution, change-sets of composite statements are larger. Therefore, the information to be stored is still considerable.

- Same as the Execution History Approach, this approach is completely supported by the debugger and is highly language dependent.

- The problem of finding accurate change-sets in the presence of pointers and composite data types still remains.

- Execution time overhead for saving change-sets is considerable.

### 2.3 How Our Approach Enhances the Existing Methods?

This thesis presents a new method for execution backtracking that overcomes the deficiencies of existing approaches. In order to implement the new execution back-
tracking method, a platform was built which integrates available research software with certain modifications. This new method is explained in detail in the following chapters of this thesis. The enhancements achieved in the new approach are as follows:

- Only the program control execution history is stored. With the assistance of a symbolic evaluator, partial change-sets can be restored while backtracking.

- By using a circular buffer, it is possible to record a partial history of program control flow. Therefore, a bounded amount of storage can be used.

- Since the amount of information to be saved is relatively small, run-time overhead is comparatively much less than the existing approaches.

- There are no constraints on backtracking. One can backtrack to any statement of the code as long as control flow information is available for that statement. For example, for loop constructs, the approach presented in this thesis allows backward execution to any iteration of the loop.

- Since backtracking is performed at the machine code level, the approach presented in this thesis is highly language independent and does not suffer from the problem of pointers and composite data type operations. The language dependency is only to the extent that it is aware of the program constructs that can alter the control flow of the program.

- Partial execution replay can be supported by this approach since the result of various instructions, even system calls, can be inferred by using reverse execution.

- The execution backtracking approach presented in this thesis is supported by the compiler, that instruments the code. Therefore, execution backtracking is
supported even for the code that fails and core dumps.

Time and space are the two important factors involved in cost estimation of any approach. In general, alternative approaches provide a span of time-space trade-offs. What makes one appropriate may depend on a specific application as well as available resources for a certain project.

A rough estimation of time and space cost for the execution backtracking approaches of Agrawal et al. [3, 2] is provided below. It is also shown that the approach presented in this thesis is less costly than Agrawal et al.'s approaches.

Consider a program with \( N \) executed program blocks and an average of \( C \) executed control statements, each program block with an average of \( S \) statements, and assume that control flow information is recorded for the whole program. In the worst case if the programmer stops at block \( N \) and backtracks to the beginning of the program, the execution time and space costs can be estimated as follows:

**Execution History Approach:** The execution time and space costs are:

- **Execution time cost:** \( O(NS) \)
- **Space cost:** program control information per statement + change-set per statement

**Structured Backtracking Approach:** The execution time and space costs are:

- **Execution time cost:** \( O(N) \)
- **Space cost:** change-set per program block

Since the change-set is recorded for each program block and not each statement, the average number of statements in each program block is not a factor in the execution time.
Our Approach: The execution time and space costs are:

- **Execution time cost**: $O(C)$
- **Space cost**: program control information per control statement

Only program control information is saved during execution time therefore, the execution time overhead depends on the number of control statements.

In general, $O(C) < O(N) < O(NS)$. This is because usually the number of control statements of a program are less than the logical blocks. In the Execution History approach, a change-set is saved for each program statement. In the Structured Backtracking approach, although fewer change-sets are recorded, the size of each change-set is huge. In our approach, not only is the amount of recorded information reduced, but also the control flow information is saved for each control statement. No record of change-sets are performed. By comparing the results shown above, it can be seen that the approach presented in this thesis is less expensive both in space and time than Agrawal et al.'s approaches to execution backtracking because the amount of information to be saved is comparatively less.

### 2.4 Summary

Software debugging is one of the most time consuming phases of software development. The debugging procedure can take more than 50% of software production time [3]. Traditional debugging tools are not adequate or efficient for diagnosing software, therefore different techniques must be developed to enhance the process of debugging. In this chapter, execution backtracking approaches by Agrawal et al. are introduced as a new technique to assist software debugging. These approaches are costly both in execution time and space. Therefore, in the following chapters of this thesis, a new
execution backtracking approach is presented that requires comparatively less space and execution time.
Chapter 3

CDB, An Operating System Independent Debugger

This chapter gives an overview of an experimental platform for implementing the execution backtracking approach presented in the thesis. The platform is an integration of existing research software with some modifications.

A machine independent C debugger, called CDB, developed by Hanson et al. [11] is used as the main debugging tool. CDB is selected for the platform because of its machine architecture independency and the simplicity of its implementation. Although CDB provides enough functionality and flexibility to debug programs for the purpose of this thesis, its implementation is simple. It consists of approximately 1500 lines of C code while GDB, the common GNU debugger, is around 150,000 lines of C [11]. In general, building a debugger for a new machine involves rewriting most of the code that deals with machine architectures and operating systems. Using CDB in this thesis allowed us to avoid the work of writing a new debugger. In addition, revising CDB is comparatively easier than revising GDB.

Target code, debugged by CDB, is generated by a retargetable C cross-compiler
(LCC) [9, 8]. In order to debug the target with CDB, only 600 lines of code were added to LCC to emit symbol table data and related code [11]. The code to be debugged has to be instrumented during compilation to support the execution backtracking approach presented in this thesis. The advantages of using LCC for this purpose are threefold. Firstly, the task of comprehending how the compiler works is trivial, especially because sufficient documentation is available regarding the design and implementation of LCC [9, 8]. Secondly, modifying LCC code is relatively easier than modifying other compilers such as GCC which do not have sufficient documentation and have a large code base and complex design. Thirdly, LCC works well with CDB, the selected debugger for this research.

Despite all the benefits of CDB stated above, modifications had to be made to CDB to make it more appropriate for the purpose of this thesis. These modifications are listed as follows. Firstly, CDB is modified to become operating system independent. Secondly, the two-process configuration of CDB, described in Section 3.1.1, is revised to run on machines with different endianness. The two major changes to CDB, stated above, are subsequently explained in Section 3.2.1.1 and 3.2.1.2 in detail.

In order to provide control over the target machine and have easier access to the target address space, a MIPS simulator, called NACHOS [5, 14], which consists of a virtual machine and operating system primitives is used. The target executable is loaded into NACHOS and is debugged using CDB debugging facilities.

In the following sections of this chapter, an overview of CDB [11] is given. Moreover, modifications made to it and its interface with NACHOS are discussed in detail.

### 3.1 Overview of CDB

In general, debuggers depend on machine architectures, operating systems, compilers and linkers. These dependencies restrict the usage of debuggers to specific platforms
with their own system software. Debugger dependencies and the reason for these dependencies are listed as follows [17]:

**Target Architecture:** different machines use different facilities to implement breakpoints and single stepping.

**Compilers and Linkers:** debuggers need to read target symbol table and object code to display source-level values and to find source-code locations.

**Operating System:** debuggers have to access the target address space and control the target execution.

CDB is a machine independent debugger which allows debugging of source level code on different hardware platforms [11]. It is a simple source level debugger for ANSI/ISO C programs compiled with LCC, a retargetable C compiler [9]. Machine independency is achieved in CDB by:

1. Embedding a part of itself, called 'a nub', in the target;

2. Using a machine independent symbol table and related debugging code, including breakpoint hooks and stopping-point data, emitted by LCC [9].

In the following subsection, the CDB design, and how the nub provides machine independency for CDB are discussed in detail.

### 3.1.1 CDB Design

The nub is the interface between the target and CDB (Figure 3.1). It is loaded with the target and defines what the debugger can do to the target or with the target. Basic functions provided by the nub are to read and write the target address space and manipulate the breakpoints and debugging information. Although the nub
provides machine independency for CDB, its implementation depends on the target machine architecture. However, since the nub interface is small, its implementation is simple. A typical nub implementation is less than 260 lines of C code [11]. Running CDB on any machine requires implementation of a new nub for that machine.

There are two different ways to run CDB [11]:

1. The first configuration, shown in Figure 3.1, is to load CDB with the target as one process. The nub is the communication channel between CDB and target. Functions that allow the debugger to access the target address space are discussed later in this chapter.

2. In the second configuration, shown in Figure 3.2, CDB runs in two different address spaces. The target is loaded with part of the nub interface, called the client, running in one address space. The rest of the nub, called the server, runs with the debugger in another address space, possibly on another host. The client is CDB’s interface to the target and provides functions to access the address space of target. The server is the part of CDB that provides the interactive environment for debugging and the command line prompt for the user. The communication between the client and server is performed using interprocess communication primitives.

The common configuration of CDB is the two-process one. The most important benefit of using this configuration is that it protects the debugger from corruption by the target program, since the debugger and target are running in different address spaces.

The program to be debugged in CDB is compiled with LCC. LCC emits the machine independent symbol table for use by the debugger. Since the symbol table information remains unchanged during program execution, CDB makes a copy of it at initialization time. When it is necessary, symbol table entries are used by CDB
Figure 3.1: One-process configuration of running CDB

Figure 3.2: Two-process configuration of running CDB
to access and display current values of the variables. In addition to symbol table information, the debugging information that is generated by LCC are breakpoint hooks and stopping-point data. Breakpoint code is injected at each stopping-point in the target at program compilation time. Stopping-point data is saved as an array at compile-time. While executing the target, breakpoint code tests stopping-point data to check if a breakpoint is set at the stopping-point. All the debugging information is added during the intermediate code generation phase of compilation. The debugging information is generated by the LCC front end, then the information is emitted by the back end of LCC. All the debugging information emitted by LCC is target independent. Both CDB and LCC are agreed on the format and representation of the data.

The trade-off of having machine architecture independency in CDB are the costs of space and time. Inserting debugging information cost a considerable amount of space. Executing the breakpoint code also extends the execution time of program. The cost of independency is roughly a factor of $3 - 4$ in both space and time [11]. In order to decrease these costs, an alternative solution is introduced by Hanson in [10]. In the revised design and implementation of CDB, debugging information is stored in external files. As a result of this approach, the space cost is reduced by nearly one-half and the time cost by $13\%$ [10]. The version of CDB used in this thesis is the one discussed in [11]. However, the revised CDB of [10] can be used to make the approach presented in the thesis more efficient and affordable. This is left for the future work.

### 3.1.2 CDB Implementation

The user interface of CDB is a simple command-line interpreter. The interface provides various debugging functions. Some of these functions are:
• set and remove breakpoints
• examine the values of variables by name
• traverse the call stack

3.1.2.1 CDB Data Structures

The data structures [11] used by the CDB’s client and server nub interfaces are as follows:

**Nub_coord.T:** describes a location in the source code where a breakpoint can be set. The location is described by a file name, a line number and a character number in that line. CDB allows the user to set breakpoints at any individual expression and at entry/exit points of compound statements.

**Nub_state.T:** describes the state of a stopped target. The target is halted by the debugger at start-up time, at breakpoints, and when faults occur. The state includes the name of a file and the source coordinate in that file where the target stopped. It has also a pointer to the stack frame and symbol table at the stopping source coordinate. These pointers are used to display the values of variables.

**module:** has the information about the stopping-points and files comprising a module. Each program consists of different modules, the module structures are linked together by the last field of their data structure.

**ssymbol:** includes all the information for each variable such as its name, type, scope, address and the file name where the variable is defined. Different symbols of each module are linked together. The type field is a pointer to another data structure which contains information about the type of identifier.

sframe: the CDB nub embeds a shadow stack in the normal call stack. The shadow stack consists of shadow frames which are created and initialized at function entry and disappear at function exit. The shadow stack is a doubly linked list of frames defined by sframe structure. This structure has information about name of the function, the module structure for the file in which the function appears and the symbol table. _Nub.tos is the address of the head of shadow stack frame. In order to fetch the value of a variable using the symbol table data, its address is calculated first. After that, the calculated address is used to access the value of the variable in the stack.

3.1.2.2 CDB Functions

The callback functions called by the nub are:

startup and fault: These two callback functions are called at the debugger initialization time. The nub state that is passed to startup includes a pointer to the symbol table before program execution begins. Using this information, the programmer can examine the initialized data and set breakpoints. The fault callback function sets up the fault handler.

Nub.set: This callback function is used for setting breakpoints at source coordinates in the source code of a program.

Nub.remove: This callback function allows the user to remove a breakpoint that has been set.

onbreak: This function is the breakpoint handler and gets invoked when execution reaches a breakpoint that has been set.
There are some other functions that are used by nub to access the target address space explained in [11]. \texttt{\_Nub\_frame} is for traversing the target stack. \texttt{\_Nub\_fetch} and \texttt{\_Nub\_store} are to read and write the number of bytes specified at a certain address from/to the target address space.

### 3.2 Revised CDB

At the beginning of this chapter, we justified the selection of the software components used to build the debugging platform for this thesis. We summarize the characteristics that motivated us to use the components below:

- **CDB**
  - an architecture independent debugger
  - small code; hence it is easy to comprehend and modify
  - possibility of running the debugger and target as two different processes which provides a more reliable debugging tool; therefore avoids debugger corruption by the target
  - sufficient debugging functionality to debug C programs

- **LCC**
  - cooperates well with CDB
  - small code and simple design makes it easier to understand and revise
  - sufficient documentation on its design and implementation makes reverse engineering more affordable and efficient
  - comparatively faster than available C compilers
  - easy to port to new targets
- NACHOS
  - provides control over execution of the target
  - easy access to target address space

Having existing software components which cooperate well with each other means that it was not necessary to write any of them from scratch. However, a considerable amount of work had to be performed to integrate these experimental tools together.

Since in the two-process configuration of CDB, the debugger and target are running as two different processes, debugger corruption by the target is avoided. Therefore, the revised version of CDB is built based on the two-process configuration. In the revised version of CDB, the nub interface to the target (client) is linked with the NACHOS code. Note that in the original version of CDB, this part is embedded in the target. In the new design, the target is loaded in NACHOS virtual memory and the debugger can access target address space by using client interface functions and NACHOS operating system primitives. The other part of CDB, which is the interface to the user (server), runs on a real operating system. The debugger and target communicate via IPC (Interprocess Communication). A high level overview of the new CDB configuration is shown in Figure 3.3.

### 3.2.1 Major Changes to CDB

The new configuration of CDB, shown in Figure 3.3, required some changes to be made to CDB. The first major change to CDB made it an operating system independent debugger. This independency was necessary in order to run CDB with the target on the NACHOS simulator, which has virtually no operating system. Although original version of CDB was machine architecture independent, it was highly dependent on the underlying operating system to support the nub. Operating system independency
adds one more system software independency to CDB which is originally a machine architecture independent debugger. The operating system independency is achieved in CDB using the system call facility of NACHOS. Unlike the original configuration of CDB where both the target and debugger machines are required to have the same endianness, the second change made to CDB allows the target and debugger processes of CDB to run on machines with different endianness. Note that in the original two-process configuration of CDB, the NACHOS/client machine and the debugger host machine had different endianness. Both of these changes to CDB provide more flexibility and portability. The details of these changes are explained in the following subsections.

3.2.1.1 Operating System Independency

In the original version of CDB, the CDB client was highly dependent on the target operating system. The client was embedded in the target running as one process on
a real operating system. Unlike real operating systems that run on bare machines, NACHOS [5, 14] runs as a single Unix process. NACHOS simulates a machine that roughly approximates the MIPS architecture. There are functions provided by NACHOS to access and manipulate its machine registers, physical memory, and virtual memory, as well as operations to run the machine or examine its state. Since NACHOS has a virtual machine, it is easy to monitor and control execution of target programs on its simulated machine.

In the revised version, the CDB client is linked with NACHOS and the target is loaded into NACHOS virtual memory. Since the nub client and target are running in two different address spaces, the target's data structures are not directly available for use by the nub client. Since the nub client is running as part of NACHOS, the system call facility of NACHOS is used by the nub interface functions to find the addresses of CDB data structures in NACHOS virtual memory.

The two key target CDB data structures, described in Section 3.1.2.1, that are used to access the debugging information and the stack are _Nub_modules and _Nub_tos. _Nub_modules is a starting point to access the information about modules, their symbol tables and stopping-point data. _Nub_tos provides access to the shadow stack frame. Two different system calls are defined in NACHOS to get the addresses of _Nub_modules and _Nub_tos from target address space at start-up time of simulator. Using these system calls makes CDB operating system independent. In the original version of CDB, the nub code was embedded with the target, therefore it was highly operating system independent.

3.2.1.2 Endianness

In the original version, the CDB host can run on a different machine than the target, but the machines must have the same endianness [11]. However, in the modified
version there is no restriction on the endianness of the target and host debugger machines. Note that NACHOS (client+target) and server machines had different endianness. A byte reordering mechanism is used for this purpose. Whenever the target and debugger machines have different endianness, the data received by the debugger will be reversed byte by byte to guarantee the correct byte ordering of the received information.

At the start-up time of the debugger, the symbol table is read from the target address space and passed to the debugger. Since this information does not change during the program execution, it is passed to the debugger only once at initialization time. If the target and host debugger machines have different endianness, the symbol table data is reversed in the debugger. During the debugging period, the user may ask for values of the variables at some target execution point. The debugger passes a request through the nub interface to the target to get the value of a variable from the address space of the target. If the target and debugger have different endianness, the value of the variable sent by target to the debugger will be reversed by the debugger.

The functions added to nub interface to fix the problem of different endianness of the target and debugger host machines are as follows:

**ReverseConstantsTable:** This function reverses the fields of the symbol table data structure at debugger initialization time.

**ReverseStypeStruct:** One of the fields of the symbol table data structure points to the type data structure that includes the information about the type of variable. This function reverses the fields of the type data structure at the start-up time of the debugger.

**ReverseString:** This function reverses the byte ordering of a simple variable. It is used by both functions mentioned above.
3.3 Summary

This chapter motivates and explains the decisions made to provide a platform for implementing a new approach to execution backtracking. It reviews the machine independent debugger, CDB, and provides the reasons why CDB is ideal for the purposes of the thesis. In addition, it also briefly introduces the compiler (LCC) and simulator (NACHOS) that collaborate with CDB. The modifications made to CDB to utilize it for this research are also explained.
Chapter 4

Execution Backtracking

This chapter presents an execution backtracking approach to program debugging. It introduces an enhanced algorithm for backtracking programs at the machine code level. Since the approach requires recording and restoration of control flow information, firstly, a brief overview of control dependence relations is provided. Then, different development phases of the approach are described in detail. Finally, the advantages and drawbacks of the execution backtracking approach are discussed.

4.1 Introduction to the Algorithm

In conventional debuggers, a user sets a breakpoint in the source code of a program and suspends program execution at that point to examine the program state (values of variables). If the value of a variable is incorrect, the user starts to think backward and test the program state at earlier stages in the program until the fault is discovered. The debugging tool developed in this thesis supports an execution backtracking approach that allows the user to reverse the direction of execution after stopping at a breakpoint in the program. By using this facility, the user avoids re-executing the
program from the beginning for each new breakpoint. In order to support the process of backtracking, only a small amount of control flow information is recorded. Unlike other approaches, no program data state is saved during program execution. Execution backtracking uses the recorded control flow information and the program's final data state (memory and register contents at the stopping point). Since the execution backtracking approach exploits the machine code representation of the program, it is able to backtrack in presence of pointers and composite data types, and is highly language independent. In order to restore the program data state to an arbitrary point of its execution, the code is executed in reverse by a MIPS R2000 symbolic reasoner [20, 12]. A bounded amount of control flow information can be saved to reduce the run-time storage cost for backtracking long lifetime programs. The trade-off is that limited backtracking will be provided.

The different phases of the execution backtracking approach are listed as follows:

1. Recording control flow
2. Recovering control flow
3. Restoring program state

Since the execution backtracking approach exploits the control flow relations of the program structure, in the following sections, firstly, the concept of program control dependence relations is discussed, and then the different phases of the execution backtracking approach are explained in detail.

4.2 Control Dependence Graph

In order to execute the program backward, some information on its execution history is required. A control dependence graph (CDG) represents the control relations
among elements of a program. It can be generated in two different ways stated as follows:

**Statically:** A static control dependence graph includes all the possible control sequences of the program. It is generated based on the program text at compile-time. Static control dependence analysis was first used by Weiser [24] in static program slicing.

**Dynamically:** It is the actual control flow for a specific program execution, and not the program text, which defines the sequence to be used. Therefore, a dynamic control dependence graph of different executions of a program vary from each other. A dynamic CDG is generated at run-time. Dynamic control dependence analysis is used by Korel et al. [13] in dynamic program slicing.

If each statement of a program is denoted by a node of the graph, and arrows among nodes indicate control dependence, the static and dynamic control dependence graphs of fragment C code of Figure 4.1 are respectively shown in Figure 4.2 and Figure 4.3. The static CDG consists of all the possible execution paths that can be taken during the program execution. The dynamic control dependence graph, shown in Figure 4.3, is for a specific execution of the example program, when input $j$ is 2. It can be observed that the dynamic CDG includes all the occurrences of a program statement during the particular execution. For example, when $j$ is 2, there are two occurrences of the `while` predicate in the dynamic CDG. In the first occurrence the `while` condition is true, therefore its body is taken. In the second iteration of the `while`, because its condition evaluates to false, the control flows to the statement after the body of the loop.

In order to record the execution history for the purposes of this thesis, dynamic control dependence is chosen over the static dependence analysis for the following reasons:
Execution backtracking is performed for a specific execution of the program, therefore the dynamic control dependence graph of that particular execution is required.

Dynamic control flow analysis is more appropriate for debugging since it is based on the run-time behavior of the program. For example, the number of iterations of a loop construct is not revealed until its run-time, therefore different occurrences of a `while` control statement are included in the dynamic CDG.

The major drawback of using a dynamic control dependence graph is that their storage space requirement is not bounded at compile-time. However, the execution backtracking approach presented in this thesis minimizes the amount of control flow information to be saved. Also to reduce the space required, a circular buffer is used to partially record the control flow information.

In the following sections, the phases of the execution backtracking approach are discussed in detail.
4.3 Recording Control Flow

Execution of program statements is not always sequential. Different control statements can change the sequence of program execution. Control statements are properties of a programming language and vary across different languages. 

Control statements can be categorized into two different classes [16]. These classes are listed as follows:

Structured: Structured control statements have simple semantics. For example, executing an if predicate may cause transfer of control to its then or else body. These two alternative transfers of control are known at compile-time.
Figure 4.3: Dynamic control dependence graph of example program 1 when input j is 2
Unstructured: These control statements have a more complicated semantics. The \textit{goto} statement is an example of this class. Control can be transferred to an address calculated at run-time.

The control dependence analysis performed in this thesis is implemented for structured control statements of the C programming language. Currently, unstructured control statements are not supported.

In order to store the program execution control sequence to build the dynamic CDG, no information is recorded unless a control statement that can cause transfer of control is reached. It should be noted that, when the program is executed sequentially and there is no transfer of control, no program control information is recorded. For example, simple assignment statements, and input and output statements do not change the control sequence of the program, therefore no information is stored for them.

In this thesis control flow recording is performed on the following C structured control statements:

• conditional statements
  
  – if/else
  
  – switch/case

• loop statements
  
  – while
  
  – do/while
  
  – for

The control statements, mentioned above, can transfer program control at execution time. In order to record program control at run-time, program code needs to be
instrumented. Modifications are made to the code generator of LCC [9] to inject the proper code for saving the control flow information of the program. Although instrumenting the code slows down the execution of the program, by using a circular buffer and minimizing the amount of recorded information, run-time overhead is reduced as much as possible. In the following subsections, the details of this instrumentation and the process of recording the control flow for each control statement are explained in detail.

4.3.1 Conditional Statements

For each if structure, less than 3 bytes are saved at run-time. The italicized statements of Figure 4.4 show how LCC instruments a C program with if structures.

The run-time data structures that are used to save the control flow information are as follows:

**if-node:** A circular buffer of if-nodes is maintained for recording the if execution data. For each if structure, an if-node is allocated in the buffer. The node consists of two different fields: 1 bit for the first field, the if-predicate-bit, indicates whether the then part of the if structure is taken or its else part; 2 bytes for the second field, the if-nesting-level, to record the nesting level of the if structure. During the control flow recovery procedure, this field is used as a key field to fetch the appropriate if-node from the circular buffer.

**if-context-stack:** Since it is possible for more than one if control statement to be active at a time, a stack of pointers, the if-context-stack, pointing to the elements of the if-nodes circular buffer is used to indicate the currently active if-node in the buffer.

**icsp:** This is a pointer to an if-node in the if-context-stack to indicate the currently
1 int i=1;
2 main()
3 {
4     create new if-predicate-bit with if-nesting-level
5     add new if-context-pointer pointing to new bit
6     if (i>1)
7         icsp=icsp-4;
8     set if-predicate-bit to one
9         i=i+1;
10        icsp=icsp+4;
11     else
12         icsp=icsp-4;
13     create new if-predicate-bit with if-nesting-level
14     add new if-context-pointer pointing to new bit
15     if (i==1)
16         icsp=icsp-4;
17     set if-predicate-bit to one
18         i=i+2;
19        icsp=icsp+4;
20     icsp=icsp+4;
21 }

Figure 4.4: Example program 2 - instrumented code
executing if control structure.

Figure 4.5 shows the run-time data structures of example program 2 of Figure 4.4, when the execution is suspended at line 18 of the program. Since the condition of the if structure of line 6 evaluates to false, the first if-bit-predicate in the circular buffer is 0, which indicates that the else body of the if structure is taken. The nesting level of this if is 1 because it is not nested within any other if structures. For the if structure of line 15, the condition is true, therefore the then part of it is executed. The if-bit-predicate of the second node of the run-time circular buffer is set to 1 which indicates that the then part of the if is executed. Note that since if structure of line 16 is nested within the if structure of line 6, its nesting level is 2. The first node of the if-stack-context is an empty node. When the program execution is not in the context of an if structure, the icsp points to this node. Since the execution of example program 2 is suspended within the context of the second if, the if context stack pointer (icsp) points to the if-node of this if.

The instrumented code performs three different sets of operations during program execution. These operations are explained below:

1. Before executing each if structure, the instrumented code performs two different operations. The first operation allocates a new if-node for the if structure.
It initializes the first field of the node to 0 and the second field to the *if* nesting level. The *if* nesting level is known at program compile-time. The second operation adds a new pointer to the *if-context-stack* which points to the allocated *if-node*.

2. After entering the context of the *if* structure, *icsp* is updated so that it points to the active *if-node*. Using this pointer, if the *then* part of the *if* structure is executed, the *if-node* bit is set to 1. If the *else* part of the *if* is executed, the bit remains 0.

3. Before exiting the *if* context, *icsp* is updated to indicate that the execution exits from the context of the *if*.

For each *switch* structure, 34 bytes are saved at run-time. The italicized statements of Figure 4.6 show the instrumentation of a program with a *switch* C structure.

The run-time data structures that are used to save the control flow information are as follows:

**switch-node**: A circular buffer of *switch-nodes* is maintained for recording the *switch* execution data. For each *switch* structure, a *switch-node* is allocated in the buffer. The node consists of two different fields: The first field, the *case-predicate-bits*, indicates which *cases* of the *switch* structure are executed at run-time. Since the maximum number of *cases* of a *switch* structure are 256 and 1 bit suffices to record the execution of a *case* statement, $256/8 = 32$ bytes are used to record the possible executions of the *cases* of a *switch* statement. The advantage of this design is that it can record the execution of more than one *case* of a *switch* structure. However, if the programmer assumes that only one *case* of a *switch* statement will be executed, less information can be saved.
int i=2;
main()
{
create new case-predicate-bits with switch-nesting-level
add new switch-context-pointer pointing to new bit
switch (i)
{
case 1:
    scsp=scsp-4;
    set first bit of case-predicate-bits to one
    i=i+1;
    scsp=scsp+4;
    case 2:
    scsp=scsp-4;
    set second bit of case-predicate-bits to one
    i=i+2;
    scsp=scsp+4;
    default:
    scsp=scsp-4;
    set third bit of case-predicate-bits to one
    i=10;
    scsp=scsp+4;
} }

Figure 4.6: Example program 3 - instrumented code
The trade-off is a matter of space and execution flexibility. The second field of the switch-node is the switch-nesting-level, which records the nesting level of the switch structure. Note that the nesting level of all cases of the switch statements are the same. This field is used in the control flow recovery procedure which is described later in this chapter.

**switch-context-stack:** Since it is possible for more than one switch control statement to be active at a time, a stack of pointers, the switch-context-stack, into the switch-nodes circular buffer is used to indicate the currently active switch-node in the buffer.

**scsp:** This is a pointer to a switch-node in the switch-context-stack to indicate the currently executing switch control structure.

The run-time data structures of the example program of Figure 4.6 are shown in Figure 4.7, when the execution is suspended at line 16. Because of the value of integer i, the second case of line 13 is executed, therefore the second bit of the case-predicate-bits is set to 1. Note that in example program 3, since there are no break statements within the case bodies, if the programmer stops execution at the last line of the program, both the second case and the default statement will be taken. Therefore, the second bit and last bit of case-predicate-bits will be set to 1. The
nesting level of the switch structure is 1 because it is not nested within any other switch statements. Since the execution is suspended within the context of the switch statement, the switch context stack pointer (scsp) is pointing to the switch-node in the circular buffer.

Execution of the instrumented code performs three different sets of operations during program execution. These operations are explained as follows:

1. Before executing each switch structure, the instrumented code performs two different operations: The first operation allocates a new switch-node for the switch structure. It initializes all the bits of the first field to 0 and the second field of it to the switch nesting level. The switch nesting level is known at compile-time. The second operation adds a new pointer to the switch-context-stack which points to the allocated switch-node.

2. After entering the context of the switch structure, scsp is updated to point to the active switch-node. During execution of the cases of the switch statement, this pointer is used to set the right bit of the case-predicate-bits.

3. Before exiting the switch context, scsp is updated to indicate that the execution exits from the context of the switch.

4.3.2 Loop Statements

For each while structure, 4 bytes are saved at run-time. Since the instrumentation procedure is same for the while, do/while and for statements, this subsection only discusses the control flow recording of while statements. The italicized statements of Figure 4.8 show the instrumentation of a program with while C structures.

The run-time data structures that are used to save the control flow information are as follows:
int i=1, j=2;
main()
{
    add a new while-node with counter initialized to zero to the buffer
    add a new while-context-pointer pointing to new node
    while (i<3)
    {
        wcsp=wcsp-4;
        increment the counter in while-node pointed by wcsp
        i++;
        add a new while-node with counter initialized to zero to the buffer
        add a new while-context-pointer pointing to new node
        while (j<4)
        {
            wcsp=wcsp-4;
            increment counter in while-node pointed by wcsp
            j++;
            wcsp=wcsp+4;
            wcsp=wcsp+4;
        }
        i=0;
        add a new while-node with counter initialized to zero to the buffer
        add a new while-context-pointer to new node
        while (i<1)
        {
            wcsp=wcsp-4;
            increment the counter in while-node pointed by wcsp
            i++;
            wcsp=wcsp+4;
        }
    }
}

Figure 4.8: Example program 4 - instrumented code
while-node: A circular buffer of while-nodes is maintained for recording the while execution data. For each while structure, a while-node is allocated in the buffer. The node consists of two different fields: The first field, the while-counter, saves the number of iterations of the while. 2 bytes for this field permits recording a maximum of $2^{16} - 1$ iterations. The second field, the while-nesting-level, records the nesting level of the while which is known at compile-time. The necessity of this field is described when discussing recovery of the control flow in the next section.

while-context-stack: Since it is possible for more than one while control statement to be active at a time, a stack of pointers, while-context-stack, into the while-nodes circular buffer is used to indicate the currently active while-node in the buffer.

wcsp: Is a pointer to while-context-stack to indicate the currently executing while control structure.

The run-time data structures of the example program of Figure 4.8 are shown in Figure 4.9, when the execution of the example program 4 is suspended at line 26. The first node of the while circular buffer indicates that the while control statement of line 6 is executed twice and its nesting level is 1 because the while is not in the

Figure 4.9: Example program 4 - run-time data structures
context of any other while. The second and third element of the while circular buffer record the number of iterations of the while of line 13 for the two different iterations of the while of line 6. Since this while is nested within another while, its while-nesting-level is 2. The execution is stopped at line 26, therefore the context of the while of line 23 is active and wcsp is pointing to the node of this while. Since the execution of the body of this while is started, its counter is incremented to 1.

Execution of the instrumented code performs three different sets of operations during program execution. These operations can be explained as follows:

1. Before executing each while structure, execution of the instrumented code, performs two different operations. The first operation allocates a new while-node for the while control statement. It initializes its counter field to 0 and its nesting level field to the nesting level of the while which is known at compile-time. The second operation adds a new pointer to the while-context-stack which points to the allocated while-node.

2. After entering the context of the while structure, wcsp is updated to point to the currently active while-node. During execution of the while body, this pointer is used to increment the while counter.

3. Before exiting the while context, wcsp is updated to indicate that the execution exits from the context of the while.

### 4.4 Recovering Control Flow

In order to recover the sequence of program counter values that occurred during program execution, a procedure has been developed that uses the control flow data recorded at execution time and control flow tables generated at compile-time. The
control flow data structures that are used to record the control flow data during program execution are explained in Section 4.3. The control flow tables that are generated at compile-time and used by the control flow recovery procedure are described below:

**while_table**: For each *while* statement, there are 3 entries to this table at compile-time. Each entry consists of a *tag* that differentiates the *while* entries, an *id* that identifies the different *while* statements, and a *label* which is a program counter (PC) value. As shown in Figure 4.10, the entries are recorded for before the *while* control statement, after the *while* control statement and the end of the *while* body.

**conditionals_table**: Entries to this table are associated with each conditional control statement. For each *if* statement, there are entries for after the *if* structure, and for the beginning and end of its *else* body. Each entry has a *tag* that differentiates entries of an *if* control statement, an *id* which is a unique identifier for each *if*, and a *label* indicating a PC value. The locations of entries are shown in Figure 4.10. For each *switch* statement, *switch* entries are recorded for the beginning of each *case*, the beginning of the *default* statement, and the end of the *switch* body. Each entry consists of a *tag* that differentiates the *switch* entries, an *id* that identifies the different *switches*, and a *label* which is a PC value. Since the first *case* determines the beginning of the *switch* structure, and the default statement is optional in *switch* statements, their *tags* are different to distinguish them from other cases.

**_control_table_**: This table provides pointers to both the *while_table* and the *conditionals_table*. The size of the tables are also recorded in this table. *_control_table_* is used by the control flow recovery procedure to search and access the entries of the *while_table* and *conditionals_table*. 

55
Figure 4.11 shows the control tables of example program of Figure 4.10. The values in the third column of the `while_table` and `conditionals_table` are resolved with the program counter values during program compilation. The label notations used in the tables show that each entry has its own PC, and entries at the same location have the same PCs. The number following the PC indicates its relative location in the code.

In order to recover the sequence of PC values that occurred at program execution time, a procedure has been developed which uses the program control flow tables generated at compile-time and the control flow data recorded at run-time. This procedure, Recovering Control Dependence Graph (RCDG), is explained in the following subsection.

### 4.4.1 Control Flow Recovery Procedure

After stopping at a breakpoint, the final PC is passed to the RCDG procedure to recover the sequence of PCs that have been executed. Starting at the final PC, if the PC does not belong to a control table entry, the PC is added to the output PC values list, and the previous PC is calculated. Since the MIPS instructions are the same length, the previous PC can be calculated by decrementing the machine instruction length from the current PC. The recovery procedure proceeds as long as run-time control flow information is available or the beginning of the program is reached (PC value is zero). If the PC belongs to a control entry in the control tables (called a special PC), the procedure performs more complicated recovery operations. These operations are based on the semantics of the control statement structure that the PC belongs to, and are discussed for each control flow statement in the following subsubsections.
```c
int i=1, j=2;

main()
{
    if (i>3) {  // if entry to conditionals table (tag='i')
        switch (i) {
            case 4:  // switch entry to conditionals table (tag='s')
                i=i+4;
                break;
            case 5:  // switch entry to conditionals table (tag='a')
                i=i+5;
                break;
            default:  // switch entry to conditionals table (tag='d')
                i=10;
                break;  // switch entry to conditionals table (tag='t')
        }
        break;  // if entry to conditionals table (tag='l')
    } else {  // if entry to conditionals table (tag='l')
        while (i<3) {  // while entry to while table (tag='b')
            i++;
            while (j<4) {  // while entry to while table (tag='c')
                j++;
            }
            i=i+j;  // while entry to while table (tag='e')
        }
    }
}
```

Figure 4.10: Example program 5 - locations of _control_table entries
### 4.4.1.1 Recovering Control Flow of Conditionals Statements

Figure 4.12 and Figure 4.13 provide the part of the RCDG pseudo code that recovers the control flow information of the *if* and *switch* control statements, respectively. As an example, consider breaking at line 23 of the example program 5 of Figure 4.10. Its *conditionals_table* is shown in Figure 4.11. The control flow recovery procedure for *if* and *switch* control statements is now described when variable *i* is initialized to 4 at the first line of the program. The run-time data structures of the program when variable *i* is 4 are shown in Figure 4.14. Since execution is stopped out of the *if* context, the *if* nesting level is 0. In the recovery procedure, first the end of the *if* structure is reached, and the *if* nesting level is incremented to 1. Using the nesting level, the associated *if-node* of this structure is fetched from the circular buffer and its *if-predicate-node* is checked. Since the bit is set to 1, it means that the *then* part of the *if* was taken, therefore the next statement to be backtracked is at line

---

**Figure 4.11: Example program 5 - _control_table**

<table>
<thead>
<tr>
<th>tag</th>
<th>id</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
<td>pc6</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>pc7</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>pc8</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>pc9</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>pc10</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>pc11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>id</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
<td>pc1</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
<td>pc2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>pc3</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>pc4</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
<td>pc5</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>pc6</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>pc11</td>
</tr>
</tbody>
</table>
initialize the if-nesting-level to 0

if (curr_pc is of an if-entry)
    if (icsp is not NULL)
        if-nesting-level=active if-node nesting level;
    else
        if-nesting-level++;

if (curr_pc is of an if_end with tag 'f')
    using if_end id get else_begin (tag='l') & if_begin (tag='i') from conditionals-table
    using the if-nesting-level find the nearest if-node with the same nesting level to the end of if circular buffer
    and check its if-predicate-bit and mark the visted if-node
    if (if-predicate-bit is 1)
        add curr_pc to PC values list & update curr_pc
        call RCDG(else_begin-4, if_begin)
        curr_pc=if_begin-4;
    else /*if-predicate-bit is 0*/
        add curr_pc to PC values list & update curr_pc
        call RCDG(if_end-4, else_begin+4)
        curr_pc=else_begin;

if (curr_pc is of an else_begin with tag 'l')
    using else_begin id get if_begin (tag='i') from conditionals-table
    add curr_pc to PC values list
    curr_pc=if_begin-4;
    if-nesting-level--;

if (curr_pc is of an if_begin with tag 'i')
    add curr_pc to PC values list & update curr_pc
    if-nesting-level--;

Figure 4.12: RCDG pseudo code for recovering if control statements
initialize the switch-nesting-level to 0

if (curr_pc is of a switch-entry)
    if (scsp is not NULL)
        switch-nesting-level=active switch-node nesting level;
    else
        switch-nesting-level++;

if (curr_pc is of a switch-end with tag 't')
    using switch-end id get all cases and default
    begin PCs from conditionals-table
    using the switch_nesting_level find the nearest switch-node
    with the same nesting level to the end of switch circular buffer
    and check switch-predicate-bits and mark the visted switch-node
    add curr_pc to PC values list & update the curr_pc
    for (each case (including first case and default statement))
        if (its bit is set)
            call RCDG(next_case_begin-4, case_begin)
    curr_pc=switch_begin-4;

if (curr_pc is of a case_begin or default_begin ( tag 's'/'a'/'d'))
    using curr_pc id get all cases and default
    begin PCs from conditionals-table
    using the switch_nesting_level find the nearest switch-node
    with the same nesting level to the end of switch circular buffer
    and check switch-predicate-bits and mark the visted switch-node
    add curr_pc to PC values list & update the curr_pc
    for (each case (including first case and default statement))
        if (its bit is set)
            call RCDG(next_case_begin-4, case_begin)
    curr_pc=switch_begin-4;
    switch-nesting-level--;

Figure 4.13: RCDG pseudo code for recovering switch control statements

60
15, the end of the then body. This statement is also the switch end, therefore the switch nesting level is updated and the appropriate switch-node is fetched from its circular buffer. From the switch-node, the procedure determines which case bodies have been taken during program execution. For the example program, the first case body is executed, therefore the PC values of the body of that case are added to the list.

4.4.1.2 Recovering Control Flow of Loop Statements

Figure 4.15 provides the part of the RCDG pseudo code used to recover the control flow information of while control statements. For example, consider stopping the execution of the example program 5 of Figure 4.10 at line 23. Its while_table is shown in Figure 4.11, and the control flow data in Figure 4.16. The control recovery procedure from line 23 to line 17 is now discussed. Since execution is stopped out of the while context, the wcsp is not pointing to any while-node, therefore the while-nesting-level is set to 0. After reaching the end of the first while at line 22, the while-nesting-level is incremented to 1 since the procedure is entering the while body. The nesting level is used to restore the number of iterations of the while by finding the most recent unmarked while-node with the same nesting level in the circular buffer. In order to find the appropriate node, the circular buffer is searched in the reverse direction of storage. After each while-node is restored, it is marked so that it is not used again. Note that when while structures are nested
initialize the while-nesting-level to 0

if (curr_pc is of a while-entry)
    if (wcsp is not NULL)
        while-nesting-level=active while-node nesting level;
    else
        while-nesting-level++;

if (curr_pc is of a while_end with tag 'e')
    using while_end id get while_control (tag='c') &
    while_begin (tag='b') from while-table
    using the while-nesting-level find the nearest while-node
    with the same nesting level to the end of while circular buffer
    and get its counter and mark the visted while-node
    if (counter is 0)
        include all PCs from while_control to while_begin in CDG
    else for (counter times)
        call RCDG(while_end-4, while_begin)
        curr_pc=while_begin-4;
        while-nesting-level--;

if (curr_pc is of a while-begin with tag 'b')
    add curr_pc to PC values list & update the curr_pc
    curr_pc=while_end;
    while-nesting-level--;

if (curr_pc is of a while_control with tag 'c')
    add curr_pc to PC values list
    curr_pc=curr_pc-4;

Figure 4.15: RCDG pseudo code for recovering while control statements
within each other, the last node of the circular buffer is not the first one visited during recovery. Therefore, the nesting level is needed to find the associated node of each while. Moreover, because the number of instances of a while counter depends on the number of iterations of its outer while, the relative position of the outer while while-node in the buffer is not known until run-time. After the counter is fetched for the while of line 17, the iterations of the while are unwound. During unwinding of the while, another while_end label is encountered which belongs to the while of line 20. The same procedure is performed to unwind this while, which includes incrementing the nesting level and fetching the proper while-node. Note that after retrieving a while-node counter, the node is marked so that it is not fetched again. Each time unwinding one iteration of the while is completed, the nesting level is decremented by 1 which means the procedure is exiting the context of that while.

4.5 Restoring Data State

The approach presented in this thesis provides limited data states restoration. In order to restore the program state to a possible point in the program execution history, the list of program counter values recovered by the RCDG procedure and program's final state are used. Since the current state and previous PC value are available, the machine instruction that was executed to obtain the current state is known. In order to obtain the previous state, the instruction should be executed in reverse. To execute
program instructions in reverse, a MIPS R2000 symbolic reasoner [20, 12] is used. For example, assume that the machine instruction `addiu $sp, $sp, 40` (add immediate unsigned) is going to be executed in reverse. During forward execution, this operation will add 40 to the stack pointer. During reverse execution, the new value of the stack pointer is known, therefore the previous value of it can be calculated. Note that, in general, the previous state remains unchanged except for the value of a memory location, or a register, the PC, and perhaps the status register, thus it is trivial to obtain the previous state. In our approach, in some cases the data restoration is impossible. For example, assume the machine instruction `le $1, 0x00000200` is going to be executed in reverse. If the prior value of $1 is coming from outside of the program (i.e. from a device), there is no way to restore register’s prior value without saving it during forward execution. However, if the prior value is derived solely from the program (i.e. initialized with a constant value), then it is possible to restore register’s prior value by continuing reverse execution. The process of restoring data state is only at a preliminary stage of investigation. Further investigation is required to determine how to best integrate saving limited data state in order to completely restore data state during reverse execution.

The program state at the machine code level consists of registers and memory values. In order to retrieve the values of variables at earlier points of program execution during debugging, a mapping procedure from high level to low level representations of variables is required. Using the standard symbol table provided for debugging, variables’ addresses in memory can be calculated and mapping can be performed easily.
4.6 Discussion

The major advantage of the execution backtracking approach presented in this thesis is that it has significantly less instrumentation overhead than existing execution backtracking approaches. Only a small amount of control flow information is recorded which makes the approach more affordable both in space and time. Unlike other approaches, no program data state is recorded during program execution. Using circular buffers, it is possible to record a bounded amount of control flow information which reduces the space cost and facilitates the process of backtracking for long lifetime programs.

Since less information is recorded during program execution, it is only possible to restore a partial record of data state on backtracking. As long as control flow information is available, backtracking can be performed. Backtracking is possible for all the iterations of the loop structures, from outside to inside and inside to outside of the control structures. In order to guarantee the correct recording of the control flow information, the minimum size of the pointer stacks to control flow circular buffers should allow the pointers to the nodes of a nested structure with the worst case nesting level to be stored in the stack. In addition, to recover the control flow information of a nested structure properly, the procedure should make sure that the nodes of all control statements of that nested structure are available in the circular buffer. If a node of a control statement of a nested structure is overwritten, the control flow information of that nested structure may not be recovered correctly. The trade-off of having more power over backtracking with less run-time instrumentation overhead is the run-time backtracking overhead.

An interesting benefit of the approach presented in this thesis is that it supports true execution replay. Since reverse execution is used to recreate partial data states of the program from the current state, partial data states of arbitrary points in pro-
gram execution can be recreated. Since the existing approaches may require forward execution of the program, the exact data states can not necessarily be recreated. This problem arises for long running programs and for programs that have complex interactions with their environment. Such interactions can be due to the use of system calls for accessing files, the network or the system console.

Since the approach exploits backtracking the program at machine code level, it is able to backtrack in presence of pointer operations. The backtracking algorithm deals with memory addresses of variables, therefore there is no overhead associated with mapping the variables from their high level to low level representation. In contrast, to overcome the problem of backtracking in the presence of pointers, other approaches require a mapping procedure which is complicated and increases the runtime overhead. Furthermore, our approach is relatively language independent since flow recording and recovery are based only on the semantics of a language’s conditional control flow constructs. In contrast, the execution backtracking approaches are highly language dependent since they are based on a language’s variable assignment and variable scope semantics which is very complicated in presence of pointers and composite data types.

Currently, the approach presented in this thesis does not support unstructured control flow. Although adding unstructured control flow will add to the control flow recording and recovery procedure overhead, our execution backtracking approach should be able to support unstructured control flow.

One major disadvantage of our approach is that it requires a procedure to execute machine instructions in reverse. A MIPS R2000 symbolic reasoner is used for executing the program backward. The problem that currently exists in the MIPS reasoner is that modifications need to be made to the SICStus constraint solving package to support reverse execution of all program instructions, especially constraints for bitwise operations. The MIPS R2000 symbolic reasoner that is used for this research
is capable of both forward and reverse execution. However, it suffices to develop a procedure that can execute programs only in reverse. Since the reverse execution of machine instructions is completely deterministic, it is possible to implement an efficient procedure for a specific machine architecture. However for complicated machine architectures, developing a procedure capable of reverse execution of machine instructions can be complex and costly.

4.7 Summary

This chapter has provided an overview of program control dependence relations. Static and dynamic control dependence graphs are briefly introduced and it is justified why dynamic dependence relations are appropriate for the purpose of debugging. An alternative execution backtracking approach and its phases are discussed in detail. The execution backtracking approach records a relatively small amount of control flow information, and is relatively more affordable than existing approaches. No record of program data states is performed except for the final state. Recording and recovering the control flow information is explained for loops and conditional control statements of the C programming language. Also, it is discussed how an arbitrary point in a program’s execution can be partially restored. In addition, the advantages and drawbacks of the approach are discussed.
Chapter 5

Conclusions and Future Work

This chapter provides a summary of the results of the work developed in this thesis. It also discusses some areas of future work that can be performed.

5.1 Summary of Results

In this thesis, an execution backtracking procedure has been developed and implemented. In order to experiment with the execution backtracking approach, a test bed has been built which integrates a number of different software with some modifications. The approach has been applied to C structured programs.

A symbolic machine independent debugger is used as the main debugging tool. The debugger is modified to be operating system independent, and to allow the two different parts of the debugger to run on machines with different endianness. A C cross-compiler is also modified to instrument the program code to be debugged.

The execution backtracking approach of this thesis is based on program dynamic control flow dependence analysis. It is developed in three different phases listed as follows:
• Recording control flow

• Recovering control flow

• Restoring program state

The execution backtracking approach presented in this thesis is superior to existing execution backtracking approaches in a number of respects; the approach records only a small amount of control flow information. Unlike other approaches, no record of program data states is performed. Limited data states can be restored. Using a circular buffer, it can be applied to save a partial record of control flow information which is useful for backtracking programs with long lifetimes. Since the amount of information to be recorded is considerably reduced, the run-time overhead both in space and time are relatively less than the existing approaches. Although the amount of recorded information is comparatively less, there are no particular constraints on backtracking. In contrast with some of the more advanced approaches, it is possible to backtrack from inside to outside and from outside to inside a program logical block. It is also capable of backtracking to different iterations of a loop control statement. When a circular buffer is used to save control flow, backtracking is possible as long as control flow information is available.

The implementations of existing execution backtracking approaches are supported by a debugger. This means that in order to perform execution backtracking, the programmer must execute the program the first time under the control of debugger. While it is feasible to instrument the code using a compiler to record the required information, the instrumentation run-time overhead in terms of space and time would be huge. Therefore, it is not practical to have support from a compiler for these approaches. The execution backtracking approach presented in this thesis is supported by a compiler. It requires a small amount of information to be recorded,
therefore its run-time overhead is relatively less. The approach is quite valuable, since it allows execution backtracking of code that fails (i.e. core dumps) in the field.

The approach exploits backtracking programs at machine code level, therefore it is able to backtrack in presence of pointers and composite data types. Since the approach deals with memory addresses of variables, there is no overhead associated with mapping the variables from their high level to low level representation. In contrast, the existing execution backtracking approaches that solve the problem of backtracking in presence of pointers require a mapping procedure which is complex and increases the run-time overhead.

In order to restore the program states, a symbolic reasoner is used to execute the machine code of a program in reverse. The symbolic reasoner uses the final program state and the list of PC values that is generated during the control flow recovery procedure to execute the program in reverse and recreate the program states partially. Since the program states that are restored are identical to those during the original execution of the program, it is possible to use the approach to support execution replay. The existing execution backtracking approaches do not support execution replay.

The existing execution backtracking approaches require the data states to be stored, therefore they are based on the semantics of a language’s variable assignment and variable scope. The approach developed in this thesis requires store and recovery of control flow information, therefore it is based on the semantics of a language’s control flow construct. Since the semantics of a language control flow constructs are simpler than the language’s data flow semantics, our approach is relatively less language dependent. Therefore, it is possible to extend the usage of the execution backtracking approach to different programming languages.
5.2 Future Work

Although the execution backtracking approach developed in this thesis enhances the existing approaches, the following tasks are suggested for further enhancements:

- Currently, the symbolic debugger that is used for the diagnosis platform of this thesis provides an interactive command prompt for the user. In order to utilize the execution backtracking approach, a more suitable user-interface (i.e. window based graphical user-interface) should be developed and integrated with the debugger.

- The execution backtracking approach developed in this thesis can be combined with dynamic code slicing to provide a more effective diagnosing tool. Currently, dynamic control flow dependence analysis is performed, which provides a subset of the program that includes all the statements of the program during a particular execution. Performing the data flow analysis on this subset can isolate the statements that have an effect on the value of a variable. The new subset is useful when the user observes unexpected behavior of a variable and needs to track down the statements that are affecting the value of that variable.

- The execution backtracking approach can be integrated with log-based rollback recovery/replay approaches for fault recovery in multi-threaded systems [7].

- The execution backtracking approach can be used to support consistency-based diagnosis [6]. Consistency-based diagnosis is an automated approach to diagnosis.

- Currently, control flow analysis of the approach is performed for structured control flow statements. Control flow analysis should be extended to cover unstructured control statements.
• Limited data states can be restored in the current approach. In order to provide complete restoration of data states, the approach can be combined with backtracking approaches that use limited data state checkpointing.

• In order to port the execution backtracking approach to a hardware platform with a specific machine architecture, it is necessary to develop an efficient procedure capable of running the program in reverse for that machine architecture. The procedure developed in this thesis only allows reverse execution of programs for a MIPS R2000 processor.
References


73


