Fault Location via Precise Dynamic Slicing

by

Xiangyu Zhang

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 0 6

Get the official approval page
from the Graduate College
before your final defense.

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: _____

Acknowledgements

First of all, I would like to sincerely thank my advisor, Dr. Rajiv Gupta. He is the person that enrolled me to the department; the person that motivated me to pursue the current career; the person that walked me through the very first paper; the person that stayed up at 4 am in the morning with me for paper deadlines; the person whom I learned a lot from, both academic and everyday.

I also greatly appreciate Dr. Neelam Gupta. Her attitude towards research has influenced me a lot.

I am very grateful to Dr. John Kececioglu and Dr. Salim Hariri for their helpful advice.

I would like to express my gratitude to the other group members: Sriraman Tallam, Youtao Zhang, Jun Yang, Bengu Li, Vijay Nagarajan, and Arvind Krishnaswamy. They have helped me a lot in these years.

I also thank my family, my wife Tiantian Qin, my mother Qiting Liang, my father Shufan Cheung, and my younger brother Zhenyu Zhang. They are the ones that always stand by me.

Finally, I would like to thank my one year old – Brian Zhang. Thank him for waking me up every 3 hours for two months.

Table of Contents

TABLE OF CONTENTS—*Continued*

Table of Contents—*Continued*

List of Figures

List of Figures—*Continued*

List of Tables

Abstract

Developing automated techniques for identifying a fault candidate set (i.e., subset of executed statements that contains the faulty code responsible for the failure during a program run), can greatly reduce the effort of debugging. Over 15 years ago precise dynamic slicing was proposed to identify a fault candidate set as consisting of all executed statements that influence the computation of an incorrect value through a chain of data and/or control dependences. However, the challenge of making precise dynamic slicing practical has not been addressed. This dissertation addresses this challenge and makes precise dynamic slicing useful for debugging realistic applications. First, the cost of computing precise dynamic slices is greatly reduced. Second, innovative ways of using precise dynamic slicing are identified to produce small failure candidate sets.

The key cause of high space and time cost of precise dynamic slicing is the very large size of dynamic dependence graphs that are constructed and traversed for computing dynamic slices. By developing a novel series of optimizations the size of the dynamic dependence graph is greatly reduced leading to a compact representation that can be rapidly traversed. Average space needed is reduced from 2 Gigabytes to 94 Megabytes for dynamic dependence graphs corresponding to executions with average lengths of 130 Million instructions. The precise dynamic slicing time is reduced from up to 20 minutes for a demand-driven algorithm to 16 seconds. A compression algorithm is developed to further reduce dependence graph sizes. The resulting representation achieves the space efficiency such that the dynamic execution history of executing a couple of billion instructions can be held in a Gigabyte of memory. To further scale precise dynamic slicing to longer program runs, a novel approach is proposed that uses checkpointing/logging to enable collection of dynamic history of only the relevant window of execution.

Classical backward dynamic slicing can often produce fault candidate sets that contain thousands of statements making the task of identifying faulty code very time consuming for the programmer. Novel techniques are proposed to improve effectiveness of dynamic slicing for fault location. The merit of these techniques lies in identifying multiple forms of dynamic slices in a failed run and then intersecting them to produce smaller fault candidate sets. Using these techniques, the fault candidate set size corresponding to the backward dynamic slice is reduced by nearly a factor of 3. A fine-grained statistical pruning technique based on value profiles is also developed and this technique reduces the sizes of backward dynamic slices by a factor of 2.5.

In conclusion, this dissertation greatly reduces the cost of precise dynamic slicing and presents techniques to improve its effectiveness for fault location.

Chapter 1
Introduction

Software is pervasive in modern society. Not only does running a business depend on computer software for production, distribution and after-sales support, but living everyday life also relies on software for communication, entertainment, and so on. Unfortunately, as Mark Paulk from Carnegie Mellon's University's Software Engineering Institute noted,"*A fundamental problem with software quality is that programmers make mistakes*" [5]. Software development is primarily a human activity and humans make mistakes. As a result, errors inevitably creep into software in spite of the advances made in the areas of programming languages and software development processes. The impact of software errors is enormous. According the report of *National Institute of Standards and Technology* (NIST) in 2002, software errors caused the US economy an estimated $59.5 billion annually, or about 0.6 percent of the *gross domestic product* (GDP). The loss arose from both the user side and the developer side. Moreover, the recent progress in computer architecture and programming languages has given rise to more and more complicated software development procedures, which in turn make software more and more vulnerable to human mistakes. Therefore, how to improve the quality of software by reducing the number of errors has posed an imminent challenge to the research community.

To improve the quality of software, both static and dynamic analyses can be used. Static analyses [10, 40, 29, 18, 76, 23, 26, 44] have the power of proving a program is free of certain types of errors. However, static analyses have limitations. First of all, they are usually accompanied by false positives, i.e. programs are identified as faulty while they are not. This is due to the conservative nature of underlying program analyses such as alias analysis. Second, static analyses are only capable of

verifying certain simple properties. Third, static analyses usually require developers to provide specifications, which many developers are reluctant to write. Therefore, static analyses cannot remove all the software errors.

A software error can manifest itself at runtime as a software failure once it has escaped detection through static analyses. In such circumstances, dynamic analyses are needed to detect and locate the error. In recent years, a wide variety of dynamic analyses have been proposed [66, 55, 69, 46, 37, 57, 35, 38, 81]. Many employ machine learning or statistical techniques to observe runtime deviations from certain invariants and raise alarms for those anomalies. Others try to understand bugs by doing search in the program state space. Such techniques produce a fault candidate set for a failed run, which is basically a set of statements that include the faulty code.

A debugging aid which finds a fault candidate set and tries to explain the cause-effect relations between faulty code and failure through dependences is called program slicing. This was first introduced by Mark Weiser [74, 75]. The program slice corresponding to a variable at a specific program point is defined to contain the subset of program statements which can potentially contribute to the computation of the value of the variable across all program executions. Weiser gave the first *static slicing* algorithm, in which the static slice is computed by taking a transitive closure over data and control dependences that directly or indirectly influence the value of the variable at a program point. Since the objective of slicing is to focus the attention of a programmer or an algorithm to a relevant subset of program statements, conservatively computed and thus usually very large static slices are undesirable in many cases.

Realizing this limitation of static slicing in debugging, Korel and Laski proposed the idea of *dynamic slicing* [49]. The dependences that are exercised during a program execution are captured precisely and saved in form of a *dynamic dependence graph*. Dynamic program slices are constructed in response to requests by traversing the captured dynamic dependence information.

The main focus of this dissertation is fault location via precise dynamic slicing. Although dynamic slicing has been invented for almost two decades and a lot of research has been carried out on various dynamic slicing algorithms [7, 11, 12, 51, 63] and also on different ways of applying dynamic slicing to fault location [48, 20, 50, 6], there remain two main challenges:

- Computation of dynamic slices is *inefficient* in terms of execution time and space.
- Dynamic slices are usually quite large limiting their *effectiveness* for fault location.

Computation of dynamic slices requires processing execution traces including the control flow trace, memory trace, and possibly value trace. The time and space requirements are in general proportional to the execution length, which makes dynamic slicing prohibitively expensive if the execution gets long. An execution of 100 Millions instruction could require up to 2 Gigabytes space, and traversing through such a high volume of dynamic information to compute a dynamic slice could take minutes. A few algorithms have been proposed to improve space efficiency [7, 12], but they also have inherent limitations. Most people still consider precise dynamic slicing as an impractical technique and most of the current implementations are only applicable to toy programs and short executions. In this dissertation, novel techniques that significantly improve the efficiency of dynamic slicing are proposed.

The second main challenge is the effectiveness problem. Classic dynamic slicing algorithms are quite effective in containing the root causes of bugs but usually produce over-sized slices which may contain thousands of source code statements. Hence, manually inspecting these slices often requires tremendous effort. Conventional algorithms consider only one type of dynamic slice of a failed run, which is the backward dynamic slice of wrong output. In this dissertation, other dynamic slices including the forward dynamic slice of failure inducing input and the bidirectional dynamic slice of

a critical predicate are identified. The combinations of these dynamic slices further reduce the fault candidate set and improve the effectiveness of dynamic slicing for fault location.

The contributions of this dissertation are summarized as follows.

- The cost and effectiveness of traditional precise dynamic slicing algorithms are thoroughly studied on realistic programs and executions. The limitations of existing techniques are identified and they serve as the motivation for this dissertation.

- A unified trace representation, *Whole Execution Trace* (WET), is designed as the solution to the *efficiency* challenge. WET efficiently captures a set of complete traces of an execution including control flow trace, value trace, and dependence trace. Sophisticated optimizations are first applied to remove redundancy in the traces so that both the time of accessing the traces and the space required to store the traces are significantly reduced. In addition, a generic compression technique, which has the novel feature of bidirectional traversibility, is further employed to reduce the space consumption. The space efficiency achieves 4 bits per executed instruction. While fault location is the only application of WET in this dissertation, it has a wide variety of other potential applications such as software security [85], compiler, and architecture research.

- One of the key observations presented in this dissertation is that traditional dynamic slicing techniques have limited effectiveness in locating faults because they consider only one type of evidence in the failed run – the wrong output. In this dissertation, new types of dynamic slices are proposed to take advantage of new types of evidences.

- Traditionally, dynamic slices are computed based on dependence profiles. This dissertation, for the first time, shows that value profiles can be used to assign

weights to each executed statement, which indicates the likelihood of that executed statement being faulty. Such likelihood estimates can be used to prune a pre-computed dynamic slice. The essence of the new technique is that edges in a dynamic dependence graph should disclose the reliability of dependences in addition to the existence of dependences. For example, if a dependence edge represents a one-to-one mapping, the correctness of the definition can be inferred from the correctness of the use, which is not true for a many-to-one mapping. This key idea and the proposed technique have the potential impact on information flow research as well.

- Finally, this dissertation also presents an effort to scale dynamic slicing to long running programs by integrating dynamic slicing with logging/replay techniques.

The rest of the dissertation is organized as follows. In chapter 2, background information is given to facilitate understanding of the remaining chapters. The challenges of efficiency and effectiveness of precise dynamic slicing are discussed in detail and the corresponding solutions are also briefly mentioned. In chapter 3, optimizations on dynamic dependence graphs, which are the basic trace representation for dynamic slicing, are described. Chapter 4 discusses how multiple types of dynamic slices can be used together to improve the effectiveness of fault location. In chapter 5, value profiles are optimized and then compressed using a novel prediction based compression technique. Chapter 6 explains how value profiles can be used to estimate the likelihood for a statement execution being faulty. This information can be used to reduce the size of a fault candidate set. Chapter 7 proposes combining tracing with checkpointing such that dynamic slicing can be scaled to much longer runs. Related work is discussed in chapter 8. Conclusions and directions for future work are given in chapter 9.

Chapter 2
The Fault Location Framework

This chapter begins by providing background information on two components of the fault location framework. It first introduces *program execution profiles* that are collected for the purpose of dynamic analysis. Second, background information on *dynamic slicing*, which is the primary form of dynamic analysis used for fault location in this work, is introduced. Next, the challenges in constructing an efficient and effective framework are identified so that the resulting framework can be used in practice for debugging real applications. This chapter also briefly describes how these challenges are addressed in the remainder of this dissertation.

2.1 Execution Profiles: The WET Representation

Profiles of different kinds have been exploited for variety of tasks such as code optimization [79, 15, 32], architecture design [22, 47, 92], debugging and testing software [7]. A comprehensive set of profile data that captures the complete functional execution history of a program run must include the following:

- *Control flow profile.* Control flow profile captures the complete control flow path taken during an execution.

- *Value profile;* This profile captures the values that are computed and referenced by each executed statement. Values may correspond to data values or addresses.

- *Dependence profile.* Dependence profile captures the information about data/control dependences exercised during an execution. A data dependence represents the flow of a value from the statement that defines it to the statement that uses it

as an operand. A control dependence between two statement represents that the execution of one statement depends on the branch outcome of a predicate in the other statement.

Together the above information tells what statements were executed and in what order (control flow profile), what operands and addresses were referenced as well as what results were produced during each statement execution (value profile), and the statement executions on which a given statement execution is data/control dependent (dependence profile).

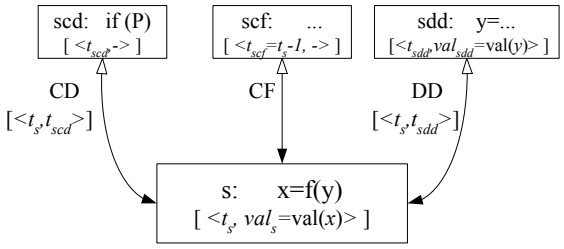
A unified representation, the Whole Execution Trace (WET), that holds a full execution history is employed in the framework. WET is essentially a static representation of the program that is labeled with the dynamic profile information. This organization provides a direct access to all of the relevant profile information associated with every execution instance of every statement. A statement in WET can correspond to a source level statement, intermediate level statement, or a machine instruction. However, in the remainder of this section, it is assumed that each statement is an intermediate code statement.

In order to represent profile information of every execution instance of every statement, it is clearly necessary to distinguish between execution instances of statements. The WET representation distinguishes between execution instances of a statement by assigning unique *timestamps* to them [89]. To generate the timestamps a *time* counter is maintained that is initialized to one and each time a basic block is executed, the current value of *time* is assigned as a timestamp to the current execution instances of all the statements within the basic block and then *time* is incremented by one. Timestamps assigned in this fashion essentially remember the ordering of all statements executed during a program execution. The notion of timestamps is also key to representing and accessing the dynamic information contained in WET.

The WET is essentially a labeled graph whose form is defined next. A label associated with a node or an edge in this graph is an ordered sequence where each element in the sequence represents a subset of profile information associated with an execution instance of a node or edge. The relative ordering of elements in the sequence corresponds to the relative ordering of the execution instances. A sequence of elements e_1, e_2,.. is denoted as $[e_1 e_2 ...]$. For ease of presentation it is assumed that each basic block contains one statement, i.e., there is one to one correspondence between statements and basic blocks.

Definition: The Whole Execution Trace (WET) is represented in form of a labeled graph $G(N, E(CF, CD, DD))$ where:

N is the set of statements in the program. Each statement $s \in N$ is labeled with a sequence of ordered pairs: $[< t_s, val_s >]$ where statement s was executed at time t_s and it produced the value val_s. Note that in general when a node contains multiple statements, instead of a single value in each ordered pair, a set of values are used, each one of which corresponds to a distinct statement in the basic block.



E is the set of edges. The edges are bidirectional so that the graph can be traversed in either direction. $(s \rightarrow d)$ denotes direction of the edge that takes us from the source s of the dependence to the destination d of the dependence while $(s \leftarrow d)$ is used to denote the reverse direction. The edges are subdivided into three disjoint categories.

- DD is the set of *data dependence* edges in the program. Each edge $(sdd \rightarrow s) \in DD$ is labeled with a sequence of ordered pairs: $[< t_s, t_{sdd} >]$ where

statement s was executed at time t_s using an operand whose value was produced by statement sdd at time t_{sdd}.

- CD is the set of *control dependence* edges in the program. Each edge $(scd \rightarrow s) \in CD$ is labeled with a sequence of ordered pairs: $[< t_s, t_{scd} >]$ where statement s was executed at time t_s as a direct result of the outcome of predicate scd executed at time t_{scd}.

- CF is the set of *control flow* edges in the program. These edges are *unlabeled*.

The example in Figure 2.1 illustrates the form of WET. A control flow graph and control flow trace of one possible execution is given in Figure 2.1a. Since the entire WET for the example is too large, the figure only shows the subgraph of WET that captures the profile information corresponding to the executions of node 8. The label on node 8 says that statement 8 is executed five times at timestamps 7, 37, 57, 77, and 97 producing values c, d, d, d, and c respectively. Executions of statement 8 are control dependent upon statement 6 and data dependent on statements 4, 2 and 15. Therefore CD and DD edges are introduced whose labels express the dependence relationships between execution instances of statements 6, 4, 2, and 15 with statement 8. Unlabeled control flow edges connect statement 8 with its predecessor 6 and successor 9 in the control flow graph.

Next it will be shown how WET can be used to respond to a variety of useful queries for subsets of profile information. The ability to respond to these queries demonstrates that the WET representation incorporates all of the control flow, data and control dependence, value, and address profile information.

Control flow path. The path taken by the program can be generated from WET using the combination of static control flow edges (CF) and the sequences of timestamps associated with nodes (N). If a node is labeled with $< t, - >$, the node that is executed next must be labeled with $< t + 1, - >$. Using this observation, the complete path or part of the program path at any execution point can be generated.

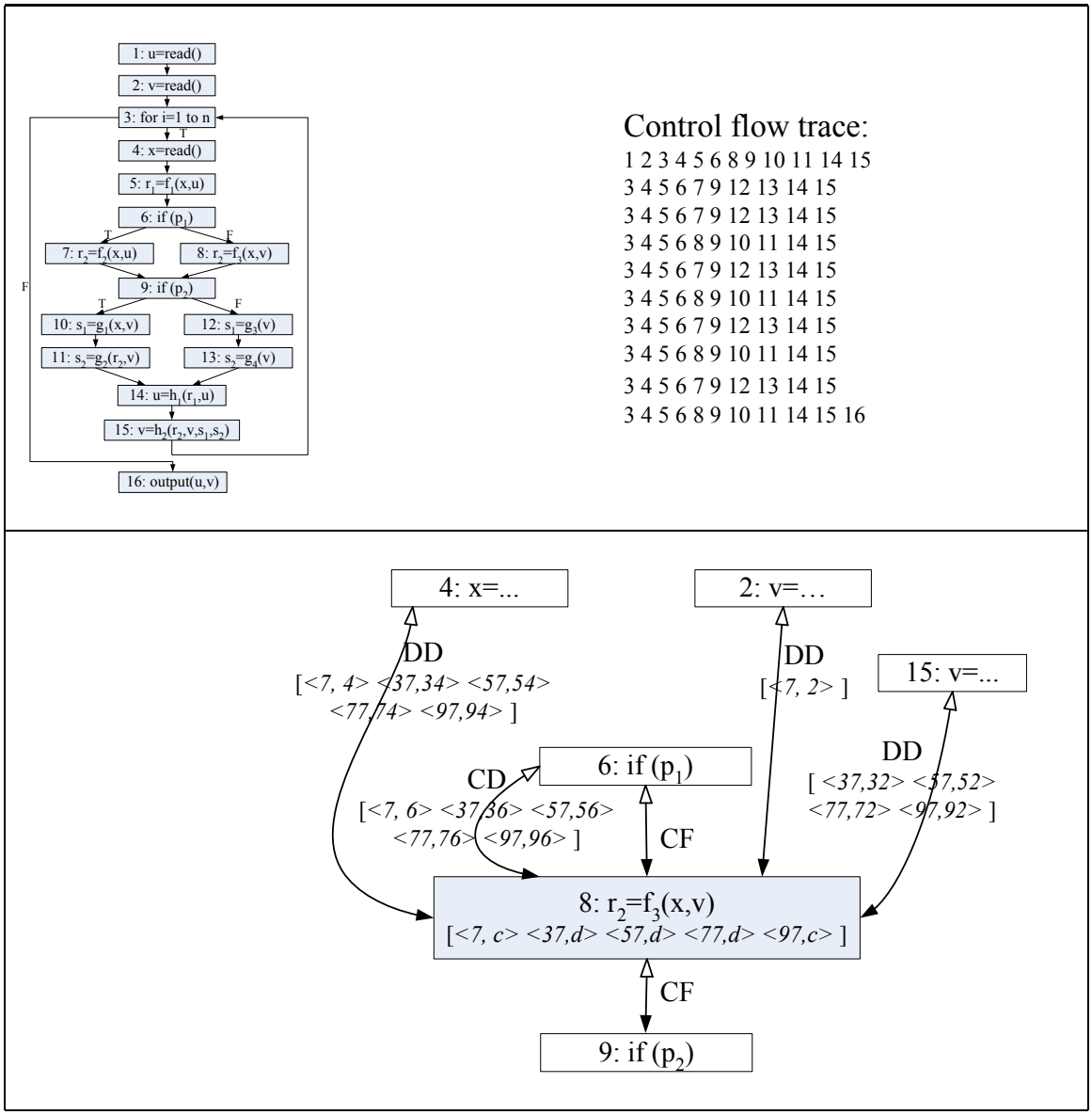


FIGURE 2.1. An example: (a) CFG and its control flow trace; (b) WET subgraph of node 8.

Values and addresses. The value and address profiles are captured by the values contained in $[< t, v >]$ sequences associated with nodes. Some values represent data while others represent addresses – the distinction can be made by examining the use of the values. Values produced by executions of a statement can be obtained by simply examining its $[< t, v >]$ sequence. Addresses corresponding to executions of a specific statement can be obtained by simply examining the $[< t, v >]$ sequences of statements that produce the operands for the statement of interest. On the other hand the sequence of values (addresses) that are produced (referenced) during program execution can be extracted by following the control flow path taken as described earlier and then examining the relevant $< t, v >$ pair of each node as it is encountered.

Data and control dependences. All instances of data and control dependences are captured explicitly by labeled edges (CD and DD). Chains of data dependences, control dependences, or combinations of both types of dependences can all be easily found by traversing the WET.

The above descriptions already explain the organization of all types of profile data in the WET representation which allows variety of queries to be responded to with ease. Given the large amounts of profile information, the sizes of WETs are expected to be extremely large. This dissertation addresses the challenge of compressing WETs in a manner that does not destroy the ease or efficiency with which queries for information can be handled.

2.2 Fault Location: Backward Dynamic Slicing

The essence of fault location is to provide programmers a fault candidate set which includes the statements that are suspected to be the root cause of a program failure. One approach for providing such a fault candidate set is *dynamic slicing* [49]. Consider a failing run which produces an incorrect output value or crashes due to dereferencing an illegal memory address. The incorrect output value or the illegal address value is

now known to be related to faulty code executed during this failed run. It should be noted that identification of an incorrect output value will require help from the user unless the correct output for the test input being considered is already available to us. The fault candidate set is constructed by computing the dynamic slice backward starting at the incorrect output value or illegal address value. Before introducing how dynamic slices are computed, the concept of *dynamic dependence graph* (DDG) is defined based on the WET representation. The nodes in a DDG are the nodes in WET without timestamp and value labels. The edges in a DDG are essentially $CD \cup DD$.

Definition 1. *Given a WET, which is a labeled graph $G(N_0, E_0(CF, CD, DD))$, the* ***Dynamic Dependence Graph*** *of a program run, $DDG(N, E)$, consists of a set of nodes N and set of directed edges E where:*

$$N = \{s \mid s[< t_s, val_s >] \in N_0\}$$

$$E = \{(s \rightarrow d)[< t_s, t_d >] \mid (s \rightarrow d)[< t_s, t_d >] \in CD \cup DD\}$$

Let $s < t_s >$ denote the execution instance of s at time t_s and $(m \rightarrow n) < t_m, t_n >$ denote a dynamic data dependence or dynamic control dependence of the execution instance of statement n at time t_m on the execution instance of statement m at t_n.

Now given an executed statement $s < t_s >$, the backward dynamic slice $DDGSlice(s < t_s >)$ is the subgraph of the DDG from which $s < t_s >$ is reachable by following the forward edges, i.e., the edges from a dependence source to a dependence destination. Given a $DDG(N, E)$, $DDGSlice(s < t_s >)$ is computed as follows:

$$DDGSlice(s < t_s >) = \{NSlice(s < t_s >), ESlice(s < t_s >\}$$

$$NSlice(s < t_s >) = \{s \cup \bigcup_{\forall (s' \rightarrow s) < t_{s'}, t_s > \in E} NSlice(s' < t_{s'} >);$$

$$ESlice(s < t_s >) = \{(s' \rightarrow s) < t_{s'}, t_s > \cup \bigcup_{\forall (s' \rightarrow s) < t_{s'}, t_s > \in E} ESlice(s' < t_{s'} >);$$

In other words, $DDGSlice(s < t_s >)$ is a labeled subgraph of the DDG, and the edge labels on $DDGSlice(s < t_s >)$ are subsets of the corresponding edge labels on the DDG. During debugging, both the statements in the slice and the dependence edges that connect them provide useful clues to the failure cause.

A more traditional definition of a dynamic slice is essentially a set of static statements, which is represented by the set of nodes in the $DDGSlice$. It is also referred to as the *dynamic backward slice* (BwS) which is given by the following equation:

$$BwS(s < t_s >) = NSlice(s < t_s >)$$

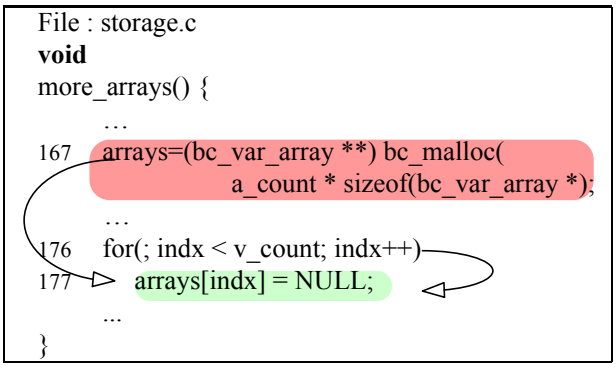


FIGURE 2.2. Heap overflow bug in $bc - 1.06$.

The benefit of backward dynamic slicing is illustrated using an example of bug in $bc - 1.06$ which causes a heap overflow error. In this program, a heap buffer is not allocated to be wide enough which causes an overflow. The code corresponding to the error is shown in Figure 2.2. The heap array *arrays* allocated at line number 167 overflows at line 177 causing the program to crash. Therefore the dynamic slice is computed starting at the address of $arrays[indx]$ that causes the segmentation fault. Since the computation of the address involves $arrays[]$ and $indx$, both statements at lines 167 and 176 are included in the dynamic slice. By examining statements at lines 167 and 176, the cause of the failure becomes evident to the programmer. It is easy to see that although a_count entries have been allocated at line 167, v_count entries

are accessed according to the loop bounds of the *for* statement at line 176. This is the cause of the heap overflow at line 177.

2.3 Challenge One: Efficient Profile Representation

Although a WET captures the full execution history of a program run, the size of WET can be very large. To demonstrate this experiments were performed. Table 2.1 lists the benchmarks considered and the lengths of the program runs which vary from 365 and 751 Million intermediate level statements. The results show that the average size of the WETs is 9589 megabytes for the execution length of 646.90 millions IR statements. As for individual components, timestamp node labels, value node labels, and edges labels consume 2467, 1731, and 5390 megabytes respectively. Note that the above WETs do not correspond to complete program runs because the ones for complete program runs were exceedingly large. As the results show, it is impossible to keep WETs in memory for the purpose of dynamic slicing especially for long program runs.

TABLE 2.1. WET sizes.

Benchmark	Input	Stmts Executed (Millions)	WET (MB)	Node labels (MB)		Edge labels (MB)
				t_s	val_s	
099.go	training	685.28	10369.32	2614.12	1849.09	5908.12
126.gcc	ref/insn-emit.i	364.80	5237.89	1391.60	1945.03	2901.26
130.li	ref	739.84	10399.06	2822.26	1894.48	5682.32
164.gzip	training	650.46	9687.88	2481.32	1733.13	5473.42
181.mcf	testing	715.16	10541.86	2728.12	1875.21	5938.54
197.parser	training	615.49	8729.88	2347.92	1615.57	4766.38
255.vortex	training/lendian	609.45	8747.64	2324.87	1641.31	4781.46
256.bzip2	training	751.26	11921.19	2865.81	2154.85	6900.52
300.twolf	training	690.39	10666.19	2633.64	1873.52	6159.03
Avg.	n/a	646.90	9588.99	2467.74	1731.13	5390.12

The Trimaran [4] compiler infrastructure was used in the above experiment. The statements here correspond to Trimaran's intermediate level statements. The programs were executed on the simulator which avoided introduction of intrusion as no instrumentation was needed. The experiments were carried out on a Pentium IV 2.4

GHz machine with 2 Gigabyte RAM and 120 Gigabyte hard disk.

Although it is desirable to hold a WET in memory, as the above experiment shows a WET may be too large. One solution to the space problem is to save the trace on disk and then traverse these traces to construct the dynamic slice on demand. In this approach instead of requiring enough memory to hold the full dynamic dependence graph, only the memory to hold the subgraph representation of the dynamic slice is needed. In [86] this approach was explored. *A demand driven analysis* on the trace was employed to recover only the relevant dynamic dependences and thus avoid constructing a full graph. When a slice computation begins the trace is traversed backwards to recover the dynamic dependences required for the slice computation. Note that since the interesting definitions will always appear earlier than the uses, a single traversal of the trace is sufficient to perform a single slice computation.

In the above algorithm, the time required to traverse a long execution trace is a significant part of the cost of slicing. This trace traversal can be speeded up as follows: the trace is divided into *trace blocks* such that each trace block is of a fixed *size*. At the end of each trace block a *summary of all downward exposed definitions* of variable names and memory addresses is generated and stored. During the backward traversal for slicing, when looking for a definition of a variable or a memory address, the algorithm first looks for its presence in the summary of downward exposed definitions. If a definition is found, the trace block is further traversed to locate the definition; otherwise using the *size* information the algorithm skips right away to the start of the trace block. Results in [86, 87] show that on average over 80% of the trace blocks are skipped.

In order to study the performance of the above demand driven algorithm, 25 slices were computed for each of program runs in the previous experiment. These slices were performed for the latest executions of 25 distinct values loaded using load statements by the program. The results are presented in Table 2.2. In the table, the sizes of the dynamic dependence graphs (DDGs) are presented. *Full* denotes the size of a full

TABLE 2.2. The performance of the *demand driven* algorithm.

Program	Executed Statements (Millions)	DDG Size (MB)		Average Slicing Time (Minutes)
		Full	Max	
099.go	138	1,707	162	10.7
130.li	125	1,745	105	11.3
126.gcc	131	1,534	58	12.1
134.perl	220	1,954	54	25.2
181.mcf	118	1,535	114	12.3
197.parser	123	1,816	40	9.9
255.vortex	108	1,442	34	10.2
256.bzip2	67	1,296	81	9.2
300.twolf	141	1,568	296	13.9
Average	130	1622	105	12.7

DDG while *Max* denotes the maximum DDG constructed during the 25 computations for each program. The average slicing time is also given. From the table, the average maximum DDG has the size of 130 megabytes while the full DDG constructed has the size of 1622 megabytes. In other words, the demand driven algorithm greatly alleviates the space problem and makes it feasible to compute dynamic slices for much longer runs. However, since computing each slice requires traversing and processing the entire trace, the slicing time is quite slow even after enabling faster traversal using trace block summaries. On average it took 9.2 to 25.2 minutes to compute a single dynamic slice across the different benchmarks. In conclusion, although the demand driven algorithm is space efficient, it is inefficient in terms of execution time.

The above discussion leads to the following challenges that must be addressed.

Challenge One: A profile representation that is both space and time efficient is desired. First, such a representation should be capable of holding a large amount of profile data in a small amount of memory. Second, this representation should be rapidly traversable so that dynamic slices can be computed in a time efficient manner.

Overview of Solutions:

- Dependence is the type of profile information required to perform conventional

dynamic slicing. A set of optimizations that greatly improve the space efficiency of a dynamic dependence graph are introduced in *chapter 3*. The optimized representation can be traversed in a time efficient manner.

- New techniques to compress value profiles are developed in *chapter 5*. As shown in chapter 6, these value profiles can be used in addition to dependence profiles to increase the effectiveness of dynamic slicing in fault location.

- A novel approach to combine checkpointing with tracing to further improve the scalability of the fault location framework is presented in *chapter 7*. Checkpointing is usually performed in an interval of minutes due to its high overhead while logging in between checkpoints can be performed with acceptable overhead. In contrast, tracing techniques can only handle an execution of a few seconds and therefore they cannot be performed for a long execution interval such as a checkpoint interval. An execution fast forwarding technique is proposed to reduce the log such that tracing becomes feasible for the replayed execution.

2.4 Challenge Two: Effective Fault Location

The effectiveness of dynamic slicing is dependent on the size of the computed fault candidate set, i.e., the size of the dynamic slice. A study was performed to evaluate the effectiveness of dynamic slicing. For the purpose of experimentation, a set of faulty versions of commonly used programs were collected. This study used real programs with real bugs that were reported by users of these programs. For programs that produced an incorrect output value, dynamic slicing was performed starting at the first incorrect output value produced during the failed run. For the programs that crashed, the value which when referenced caused the crash served as the basis for computing the dynamic slice. The faulty versions of the programs along with the descriptions of the faults are given in Table 2.3. The sources of these faulty versions

are also given. As shown in the table, these programs are widely used. In addition one should note that the first nine faults cause the programs to produce wrong outputs while the last seven faults contain memory bugs leading to a segmentation error.

Now lets see how dynamic slicing reduces the amount of code the programmer has to examine to locate faulty code. In Table 2.4, *LOC* is the lines of code in each program, *Exec* represents the lines of code that were actually executed during the failed run. *Slice* gives the sizes of the dynamic slices.

From the experimental data presented in Table 2.4, the following observations can be made:

- *Oversized Slices.* As shown by Table 2.4, although dynamic slices significantly reduce the sizes of the fault candidate sets – the sizes of dynamic slices range from 0.90% to 63.18% of the executed statements while only 1.45% to 15.58% of code was executed, the raw sizes can still be quite large. Some computed slices contain over one thousand statements. Therefore, it would be quite tedious to manually inspect these slices.

- *Applicability.* Dynamic slicing was not applicable in all the faults studied. For the first four bugs of the `grep` program, because the failed runs did not produce any output, which was essentially the misbehavior, erroneous values could not be identified on which dynamic slices could be computed.

Therefore, the following issues need to be addressed in order to deliver a highly effective fault location system.

Challenge Two: Additional types of dynamic slices must be identified so that applicability of dynamic slicing can be broadened to wider range of situations. Moreover, by using different kinds of dynamic slicing in conjunction smaller fault candidate set must be produced.

TABLE 2.3. Faults used in the study.

Program	Bug Description	Source
grep 2.5	using -i -o together produces wrong output	http://savannah.gnu.org
grep 2.5.1	(a) using -F -w together produces wrong output	http://savannah.gnu.org
	(b) using -o -n together produces wrong output	http://comments.gmane.org/gmane.comp.gnu.grep.bugs/
	(c) "echo dofe — grep dofe" finds no match	http://comments.gmane.org/gmane.comp.gnu.grep.bugs/
flex 2.5.31	(a) some variable is not defined with option -l, which fails the compilation of xfree86	http://soureforge.net
	(b) string "]]" is not allowed in user's code	http://soureforge.net
	(c) the generated code contains extra #endif	http://soureforge.net
make 3.80	(a) backslashes in dependency names are not removed	http://savannah.gnu.org
	(b) fail to recognize the updated file status while there are multiple target in the pattern rule	http://savannah.gnu.org
gzip-1.2.4	1024 byte long filename overflows into global variable	AccMon [91]
ncompress-4.2.4	1024 byte long filename corrupts stack return address	AccMon [91]
polymorph-0.4.0	2048 byte long filename corrupts stack return address	AccMon [91]
tar-1.13.25	wrong loop bounds lead to heap object overflow	AccMon [91]
bc-1.06	misuse of bounds variable corrupts heap objects	AccMon [91]
tidy-34132	memory corruption problem	AccMon [91]
mutt-1.4.2.1i	heap buffer bound miscalculation	http://www.securiteam.com/

TABLE 2.4. Sizes of backward dynamic slices.

Program	LOC	Exec (LOC%)	Slice (Exec%)
grep 2.5	8581	1157 (13.48%)	-
grep 2.5.1 (a)	8587	509 (5.93%)	-
grep 2.5.1 (b)	8587	1123 (13.08%)	-
grep 2.5.1 (c)	8587	1338 (15.58%)	-
flex 2.5.31 (a)	26754	1871 (6.99%)	695 (37.15%)
flex 2.5.31 (b)	26754	2198 (8.22%)	272 (12.37%)
flex 2.5.31 (c)	26754	2053 (7.67%)	50 (2.44%)
make 3.80 (a)	29978	2277 (7.60%)	981 (43.08%)
make 3.80 (b)	29978	2740 (9.14%)	1290 (47.08%)
gzip-1.2.4	8164	118 (1.45%)	34 (28.81%)
ncompress-4.2.4	1923	59 (3.07%)	18 (30.51%)
polymorph-0.4.0	716	45 (6.29%)	21 (46.67%)
tar-1.13.25	25854	445 (1.72%)	105 (23.60%)
bc-1.06	8288	636 (7.67%)	204 (32.07%)
tidy-34132	31132	1519 (4.88%)	554 (36.47%)
mutt-1.4.2.1	71774	2551 (3.55%)	1052 (41.24%)

Overview of Solutions:

- Two new types of dynamic slices, *forward* and *bidirectional* slices, are introduced in *chapter 4*. They greatly compensate for the limitations of the *backward* dynamic slices. A coarse grained reduction on the sizes of fault candidate sets can be achieved by intersecting multiple types of slices.

- A fine grained fault candidate set reduction technique using value profiles is presented in *chapter 6*. By looking at the correct output produced in a faulty run and the values computed during the execution, it can be inferred that many statement executions in a slice are free of errors such that they can be eliminated from the fault candidate set.

Chapter 3
Efficiency of Dependence Profiles

A dynamic slicing algorithm would be cost effective if the dynamic dependence graphs could be compacted so that they are small enough to hold in memory and the design of the compacted graphs is such that they can be rapidly traversed to compute dynamic slices. An optimization algorithm is devised to eliminate redundancy from a DDG to achieve both space and time efficiency. The algorithm is based upon the following key ideas.

Sharing a dependence edge across multiple dynamic instances: In general, it is not merely sufficient to remember whether a pair of statements was involved in a dynamic (data or control) dependence. For computing dynamic slices it is also necessary to remember the specific execution instances of the statements that are involved. Conditions are identified under which it is not necessary to remember the execution instances of statements involved in a dependence. Thus, a single representative edge can be shared across all dynamic instances of an exercised dependence. In particular, in these situations there is a one to one correspondence between *all* execution instances of a pair of statements involved in a dependence because the statements involved are *local* to the same basic block. In presence of *aliasing*, multiple definitions of a variable may reach a use even if the definitions and use are local to a basic block. In such situations *partial* sharing is possible.

Transformations for increasing sharing: It is possible to construct a transformed dynamic dependence graph in a manner that converts *non-local* dependence edges across basic blocks into *local* dependence edges and therefore increases the

amount of sharing. First, in some situations *non-local def-use* dependence edges can be replaced by *local use-use* edges. Second, *non-local def-use* dependence edges can be converted into *local def-use* dependence edges by performing *path specialization*. To limit the increase in static code size due to path specialization, this transformation is applied selectively in a *profile guided* fashion. In particular, selected intraprocedural Ball Larus paths are specialized [9].

Removing redundancy labels on non-local edges: There are situations in which different dependence edges are guaranteed to have identical timestamp pair labels. Redundant copies of timestamp pairs can thus be discarded.

The experimental evaluation shows that once sharing of edges is achieved, the number of dynamic dependence edges is reduced to roughly 6% of total dynamic edges. When the full graph sizes range from 0.8 to 1.95 Gigabytes in size, the corresponding compacted graphs range from 20 to 210 Megabytes in size. Average slicing times for the algorithm range from 1.74 to 36.25 seconds across the benchmarks studied while average slicing times of a demand driven algorithm range from 4.69 to 25.21 minutes.

In the remainder of this chapter, a series of optimizations that reduce the number of labels that need to be stored in a DDG. Once these optimizations are presented, it will also become clear that after optimization, the traversal of edges can be carried out in a much more time efficient manner.

3.1 Optimizations on Data Dependence (DD) edges

In this section, optimizations on DD edges are discussed. The subgraph constructed by the DD edges is also referred to as *the dynamic data dependence subgraph* (dyn-DDG). Given an execution instance of a use $u < t_u >$, during dynamic slicing, there is

a need to find the corresponding execution instance of the relevant definition $d < t_d >$. There are two steps to this process: (finding d) in general many different definitions may reach the use but it is necessary to find the relevant definition for $u < t_u >$; and (finding t_d) even if the relevant definition d is known it is needed to find the execution instance of d, i.e. $d < t_d >$, that computes the value used by $u < t_u >$. While in general it is necessary to remember all dynamic instances of all dependences, next it is shown that all dynamic instances need not be remembered explicitly. It is possible to infer some of the dynamic data dependences and their timestamps.

3.1.1 (OPT-1) Infer

(OPT-1a) Infer Local Def-Use for Full Elimination. Consider a definition d and a use u that are *local* to the same basic block, d appears before u, and there is no definition between d and u that can ever prevent d from reaching u. In this case there is a one-to-one correspondence between execution instances of d and u. Since d and u belong to the same basic block, the timestamps of corresponding instances are *always the same*, i.e. given a dynamic data dependence $(d \rightarrow u) < t_d, t_u >$ it is always the case that $t_d = t_u$. Therefore, given the use instance $u < t_u >$, the corresponding d is known statically and the corresponding execution instance is simply $d < t_d >$. Thus there is no need to remember dynamic instances individually – it is enough to introduce a static edge from u to d.

As mentioned earlier, in the plain WET representation, the dynamic dependence graph is collected by starting with a set of nodes (basic blocks) and then introducing all dependence edges dynamically. To take advantage of the above optimization an edge is simply introduced from u to d statically prior to program execution. No new information will be collected or added at runtime for the use u as the edge from u to d does not need any timestamp labels. In other words all dynamic instances of def-use edge from u to d are statically replaced by a single shared representative edge.

The impact of this optimization is illustrated using the example in Figure 3.1.

As shown in Figure 3.1, basic block 1 contains a labeled local def-use edge which is replaced by a static edge that need not be labeled by this optimization. Static edges are depicted as dashed edges to distinguish them from dynamic edges.

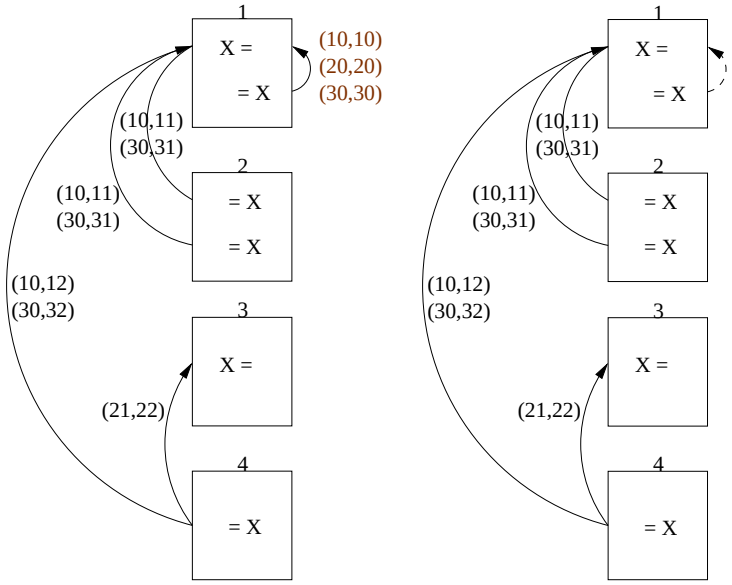


FIGURE 3.1. Effect of applying OPT-1a.

(OPT-1b) Infer Local Def-Use for Partial Elimination. In the above optimization it was important that certain subpath was free of definitions of the variable involved (say v) so that a dependence edge involving v that is free of labels could be used. In programs with pointers, the presence of a definition of a *may alias* of v may prevent us from applying the optimization even though at runtime this definition may rarely redefine v. To enable the application of preceding optimization in presence of definitions of may aliases of v we proceed as follows. A static unlabeled edge is introduced from one definition to its potential use. If at runtime another may alias turns out to truly refer to v, additional dynamic edges labeled with timestamp pairs will be added. The effect of this optimization is that the timestamp labels corresponding to the statically introduced data dependence are eliminated while the labels for the

dynamically introduced data dependence edge are not, i.e. labels have been *partially eliminated*.

During traversal, first the labels on dynamic edges are examined to locate the relevant dependence. If the relevant dependence is not found, then it must be the case that the dependence involved corresponds to the static edge which can then be traversed. It should also be clear that greater benefits will result from this optimization if the edge being converted to an unlabeled edge is the more *frequently exercised* dependence edge. Thus, if *profile data* is available it can be used in applying this optimization.

In the example shown in Figure 3.2 let us assume that $*P$ is a may alias of X and $*Q$ is a may alias of Y. Further assume that the code fragment is executed twice resulting in the introduction of the following labeled dynamic edges: between the uses of X and definitions of X and $*P$; and between the uses of Y and the definitions of Y and $*Q$. The following static unlabeled edges are introduced: from the use of X to the definition of X (as in OPT-1a); and later the use of Y to the earlier use of Y (as in OPT-2b described later). The dynamic edges introduced are: from the use of X to the definition of $*P$; and from the later use of Y to the definition of $*Q$. Thus some, but not all, labels have been removed.

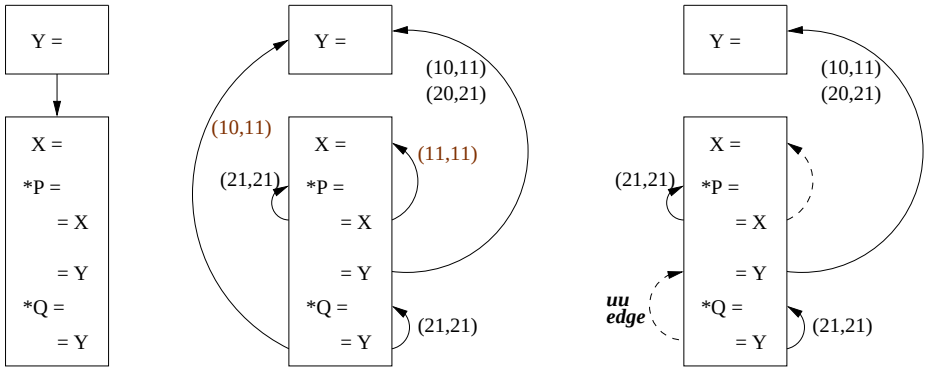


FIGURE 3.2. Effect of applying OPT-1b.

3.1.2 (OPT-2) Transform

(OPT-2a) Transform Local Def-Use for Full Elimination. While the above optimization was able to achieve partial elimination of labels, next an optimization is presented that can eliminate all of the labels present in situations with aliasing. Full elimination of labels is achieved through *specialization*. Given a use of variable v in a node (basic block) that is reachable by two distinct definitions (say d_1 and d_2) that may define v, two copies of the node are created. One copy is used to exclusively represent dynamic dependences between d_1 and the use of v while the other copy is used to represent only the dynamic dependences between d_2 and use of v. Since in each copy of the node the use of v is always data dependent upon the same definition point of v, the timestamp labels on these edges do not need to be maintained.

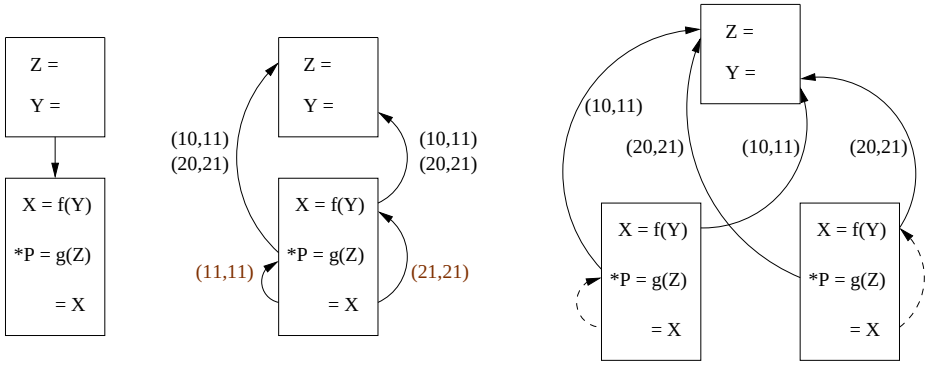


FIGURE 3.3. Effect of applying OPT-2a.

Consider the example shown in Figure 3.3. One use of X is reached by the definition of X in statement $X = f(Y)$ while the second use of X is reached by the definition of X in statement $*P = g(Z)$. By making two copies of the basic block that contains the two definitions and the use, static edges can be introduced to represent both of the above dependences and thus the labels corresponding to these edges are eliminated. Note that the dependence edges corresponding to the uses of Y and Z in the basic block must also be replicated and appropriately labeled.

In the above example, two copies of the node were sufficient to eliminate the

local labels. In general, if uses of multiple variables have multiple definitions due to aliasing, greater number of copies will be required to be created to eliminate all of the local labels. If the list of labels is very long, node replication may be justified. However, if there are only few labels, partial elimination may be preferable to full elimination.

Since the above optimizations show that timestamp labels on local dependences edges can be eliminated, optimizations that convert non-local dependence edges into local dependence edges are further developed. Once non-local dependence edges have been converted to local dependence edges, their labels can be eliminated using the above optimizations.

(OPT-2b) Transform Non-Local Def-Use to Local Use-Use. Consider two uses u_1 and u_2 such that u_1 and u_2 are *local* to the same basic block, u_1 and u_2 always refer to the same location during any execution of the basic block, and there is no definition between u_1 and u_2 that can cause the uses to see different values. Now let us assume that a non-local definition d reaches the uses u_1 and u_2. In this case each time u_1 and u_2 are executed, two non-local def-use edges $(d \rightarrow u_1) < t_d, t_{u_1} >$ and $(d \rightarrow u_2) < t_d, t_{u_2} >$ are introduced. Let u_1 appear before u_2. The non-local def-use edge $(d \rightarrow u_2) < t_d, t_{u_2})$ can be replaced by a local use-use edge $u_1 \rightarrow u_2$. The latter does not require a timestamp label because t_{u_1} is always equal to t_{u_2}. By replacing a non-local def-use edge by a local use-use edge, labels on the edge are eliminated. During slicing an extra edge (the use-use edge) will be traversed. Moreover, use-use edges are treated differently. In particular, a statement visited by traversing a use-use edge is not included in the dynamic slice.

Using static analysis one can identify uses local to basic blocks which always share the same reaching definition. Once having identified these uses, use-use edges are statically introduced from later uses to the earliest use in the basic blocks. After having introduced these edges, there will not be any need to collect or introduce any

dynamic information corresponding to the later uses.

The impact of this optimization is illustrated by further optimizing the dyDDG obtained by applying OPT-1a. As shown in Figure 3.4, basic block 2 contains a two uses of X each having the same reaching definition from block 1. The labeled non-local def-use edge from the second use to the definition is replaced by an unlabeled static use-use edge by this optimization. A use-use edge is represented by a dashed edge to indicate it is static and further indicate that it is a use-use edge.

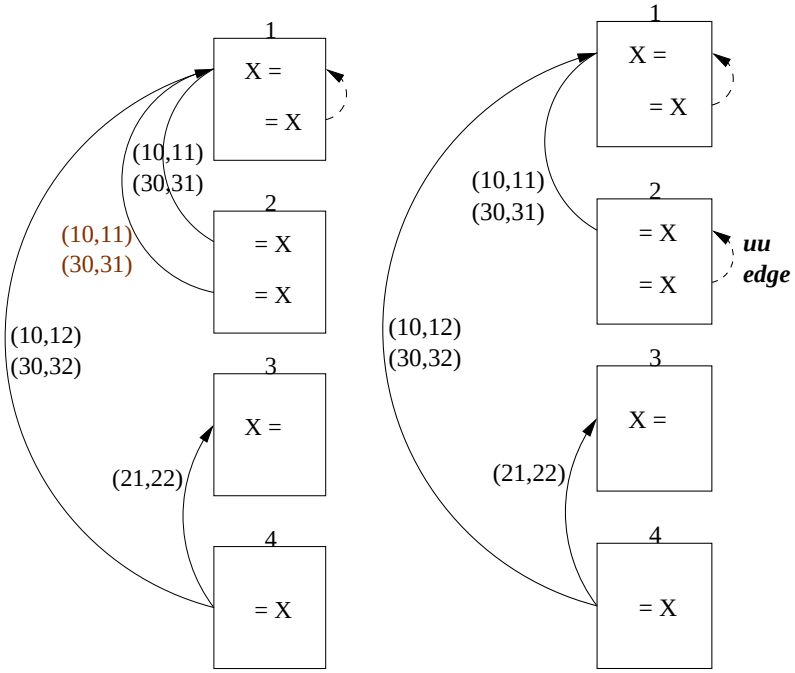


FIGURE 3.4. Effect of applying OPT-2b.

(OPT-2c) Transform Non-Local Def-Use to Local Def-Use. Given non-local def-use edge $(d \rightarrow u) < t_d, t_u >$ between basic blocks b_d and b_u, by creating a specialized node for the *path* (say p) that when executed *always establishes* the def-use edge $(d \rightarrow u) < t_d, t_u >$ (i.e., d cannot be killed along p prior to reaching u), this non-local dynamic edge can be converted into a local dynamic edge $(d \rightarrow u) < t'_d, t'_u >$ for path p. While for the original edge $(d \rightarrow u) < t_d, t_u >$ the values of t_d and t_u

are not equal, for the modified edge $(d \rightarrow u) < t'_d, t'_u >$ the values of t'_d and t'_u are equal. At runtime if the dependence between d and u is established along path p, then that dependence would be represented by an unlabeled edge local to node for path p. However, if the dependence is established along some path other than p, it is represented using a labeled non-local edge between b_d and b_u.

The consequence of earlier optimizations was that the initial graph contains some statically introduced data dependence edges. The consequence of this optimization is that instead of containing only basic block nodes, the graph contains additional nodes corresponding to paths that have been specialized. During execution it must be detected when specialized paths are executed (an algorithm to do so is presented later). This is necessary for construction of the DDG due to the following reasons. The value of global timestamps must be incremented after the execution of code corresponding to a node in the graph. Thus, the timestamp will no longer be incremented each time a basic block is executed because nodes representing specialized paths contain multiple basic blocks. At runtime the system must distinguish between executions of a block that correspond to its appearance in a specialized path from the rest of its executions so that when a dynamic data dependence edge is introduced in the graph it is known which copy of the block to consider.

The impact of this optimization is illustrated by further optimizing the optimized dyDDG from Figure 3.4. As shown in Figure 3.5, if a specialized node is created for path along basic blocks 1, 2 and 4, many of the previously dynamic non-local def-use edges are converted to dynamic local def-use edges within this path. The def-use edges established along this path can now be statically introduced within the statically created node representing this path. Thus, the timestamp labels for these def-use edges are no longer required. Since block 2 can only be executed when path 1-2-4 is executed, it is not needed to maintain a separate node for 2 once node for path 1-2-4 has been created. However, the same is not true for blocks 1 and 4. Therefore nodes representing them are maintained to capture dynamic dependences that are

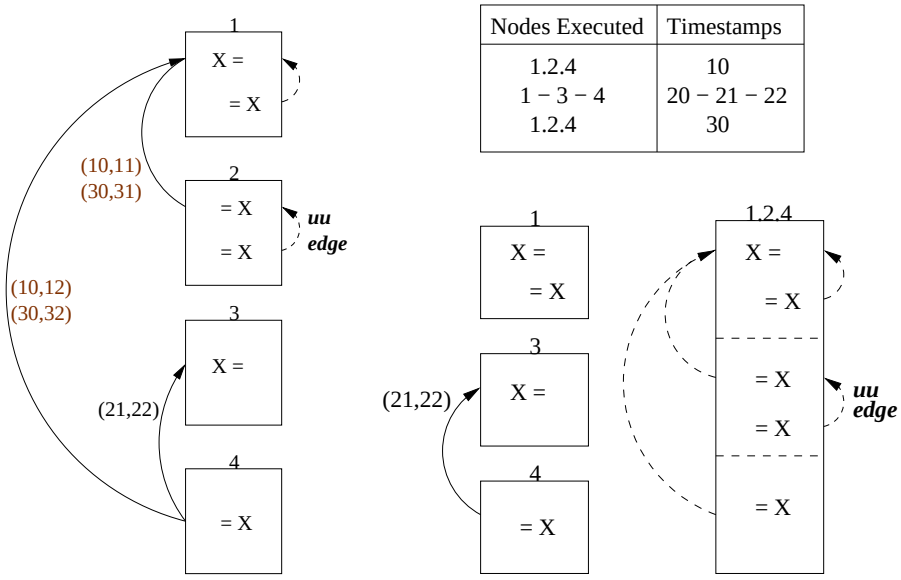


FIGURE 3.5. Effect of applying OPT-2c.

exercised when path 1-2-4 is not followed.

After applying multiple optimizations to the dyDDG of Figure 3.1(a), all but one of the labels in the dyDDG have been eliminated. In fact this label can also be eliminated by creating another specialized node for path containing blocks 3 and 4.

Finally it should be noted that the above optimization only eliminates labels corresponding to dependence instances exercised along the path for which a specialized node is created. Thus, greater benefits will be derived if the path specialized is a *frequently executed path*. As a result, selection of paths for specialization can be based upon *profile data*.

3.1.3 (OPT-3) Redundancy Across Non-Local Def-Use Edges

In all the optimizations considered so far, situations have been identified and created in which the labels were guaranteed to have a pair of identical timestamps. Now an optimization is presented which identifies pairs of dynamic edges between different statements that are guaranteed to have identical labels in all executions. Thus, the

statements can be clustered so that they can share the same edge and thus a single copy of the list of labels. Given basic blocks b_d and b_u such that definitions d_1 and d_2 in b_d have corresponding uses u_1 and u_2 in b_u. If it is guaranteed that along every path from b_d to b_u either both d_1 and d_2 will reach u_1 and u_2 or neither d_1 nor d_2 will reach u_1 and u_2, then the labels on the def-use edges $d_1 \rightarrow u_1$ and $d_2 \rightarrow u_2$ will always be identical. The example in Figure 3.6 shows that the uses of Y and X always get their definitions from the same block and thus dependence edges for Y and X can share the labels. A shared edge between clusters of statements (shown by dashed boxes) is introduced by this optimization.

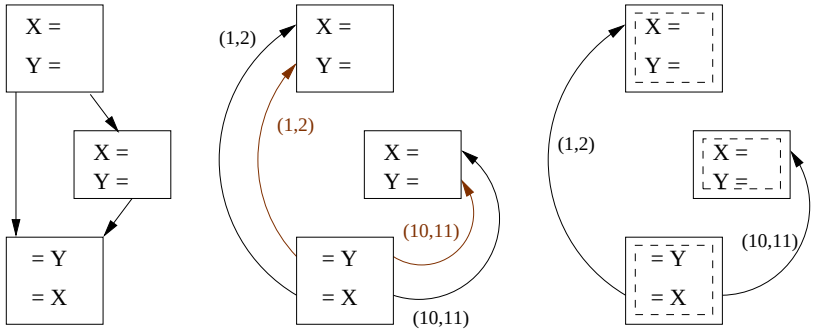


FIGURE 3.6. Effect of applying OPT-3.

3.2 Optimizations on Control Dependence Edges

Control dependences are introduced at the granularity of basic blocks. Next the optimizations that enable introduction of static unlabeled control dependence edges are presented. The subgraph of a DDG which consists of only control dependence edges is also referred to the dynamic control dependence graph (dyCDG).

3.2.1 (OPT-4) Infer Fixed Distance Unique Control Ancestor.

Often basic blocks (nodes) in a control flow graph have a unique control ancestor. Whenever a node is control dependent upon a unique conditional predicate, the con-

trol dependence edge can be introduced statically. In addition, sometimes the difference in the timestamps corresponding to a dynamic control dependence is a compile time constant. Thus, the difference value can be remembered and labeling the edge with a timestamp pair can be avoided each time the dependence is exercised. In particular, for a dynamic control dependence edge $(c \rightarrow d) < t_c, t_d >$ which satisfies the above conditions, $t_c + \delta = t_d$ because timestamp is incremented by δ whenever after the execution of the predicate when control transfers to the dependent basic block. When this optimization is applied to the example from Figure 3.7, the δ values of edges from node 2 to node 1 and node 4 to node 2 are determined to be the value 1.

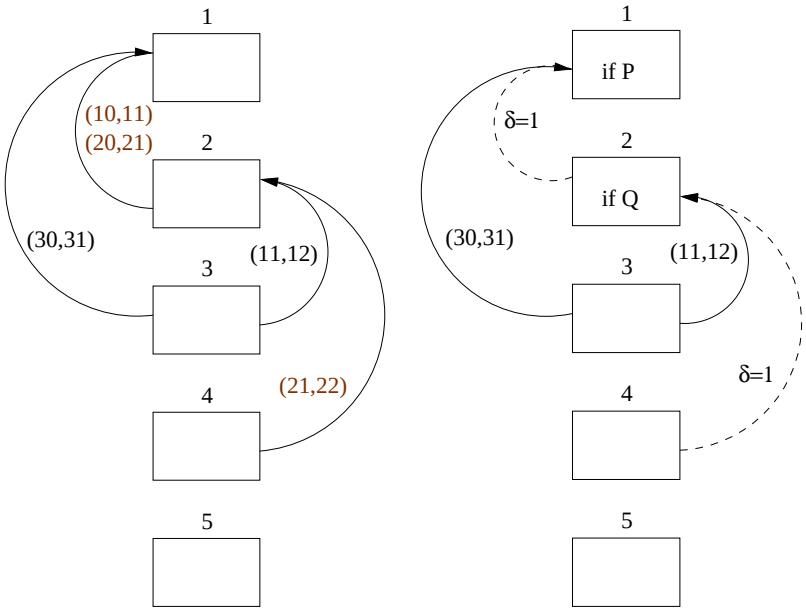


FIGURE 3.7. Effect of applying OPT-4.

3.2.2 (OPT-5) Transform

(OPT-5a) Transform Multiple Control Ancestors If a node has multiple control ancestors, the node creating specialized copies can be replicated for each of the control ancestors. Static control dependence edges can now be introduced and their δ values can be remembered. The dynamic timestamp labels are no longer required. Continuing with the example from Figure 3.7, the labeled edges corresponding to the

two control ancestors of node 3 can be replaced by static edges after replicating 3 as shown in Figure 3.8.

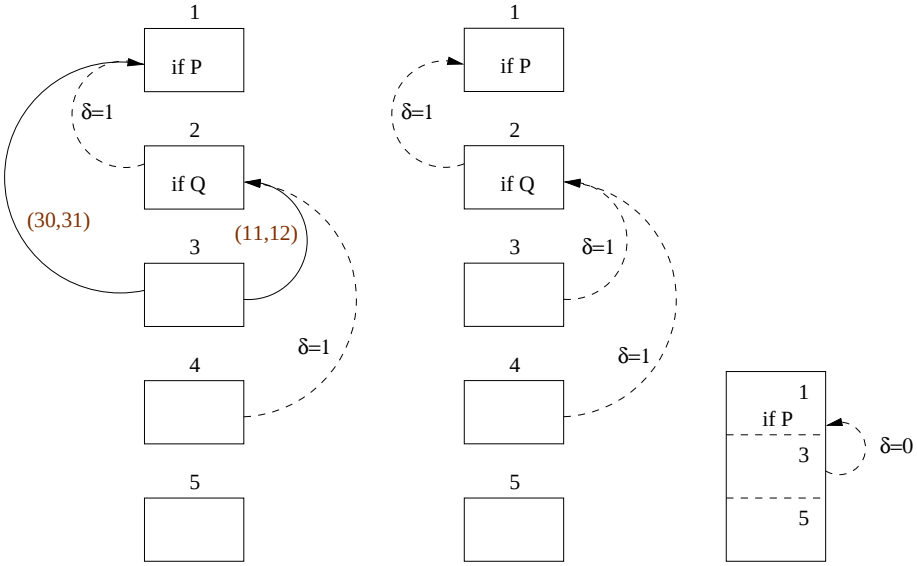


FIGURE 3.8. Effect of applying OPT-5a.

Specialization also enables another optimization for control dependences which is analogous to OPT-2b. Following specialization, a node representing a path may contain multiple basic blocks that are control equivalent [28]. Instead of using separate non-local edges for two control equivalent blocks, the non-local edge for the second block can be replaced by a local edge which points to the first block.

(OPT-5b) Transform Varying Distance Unique Control Ancestor. In optimization OPT-4 it had been shown how to handle the case when a node had a unique control ancestor which was at a constant distance from the node. It is possible that there are multiple paths from the control ancestor to the control dependent node causing the former to be at varying distances from the latter depending upon the path taken. In this case specialization can be applied to create copies of the dependent node such that each copy created is at a constant distance from the control ancestor.

In Figure 3.9, node 4 is at distance 3 from node 1 along path 1.2.3.4 and at

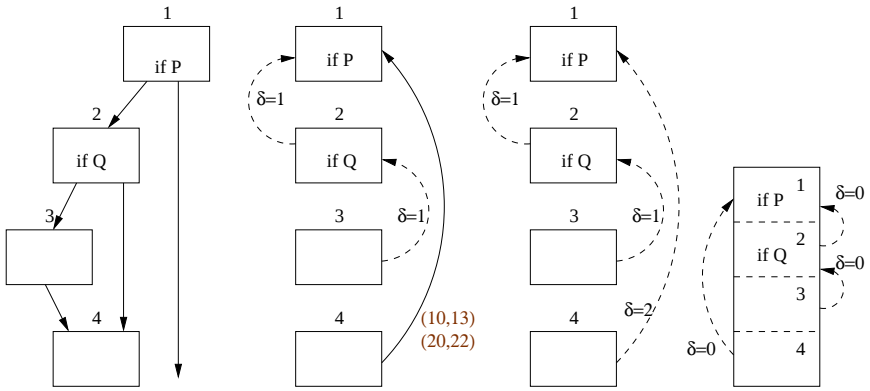


FIGURE 3.9. Effect of applying OPT-5b.

distance 2 from node 1 along path 1.2.4. By specializing path 1.2.3.4 as shown in the figure the control dependence edge from 4 to 1 can be converted into a pair of control dependence edges that are each at constant distances of 2 and 0.

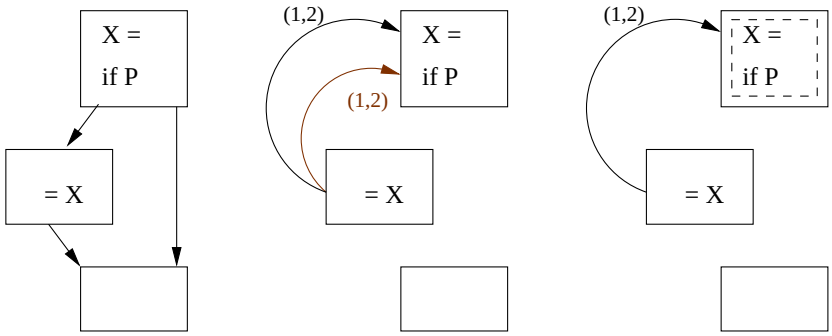


FIGURE 3.10. Effect of applying OPT-6.

3.2.3 (OPT-6) Redundancy Across Non-Local Def-Use and Control Dependence Edges

In OPT-3 it was shown how two non-local data dependence edges can share common labels. The same approach can be extended to allow a non-local control dependence edge to share labels with a non-local data dependence edge as long as these edges connect the same pair of blocks. An example illustrating this optimization is shown in Figure 3.10.

3.2.4 Completeness of the Optimization Set

In an unoptimized DDG any dependence edge may have a long list of labels attached to it. To compact the graph it is desirable to apply transformations that can eliminate this list of labels. Given this requirement, it is important that an optimization (or a series of optimizations) is available that can eliminate any list of labels. A set of optimizations can be considered to be *complete* if for any given list of labels, a sequence of optimizations can be found in the optimization set that can be used to eliminate the list of labels. The *completeness* property of the optimization set is important because it indicates that there are sufficient optimizations and it is not needed to continue developing additional ones. In fact given an optimization set that is complete, it is possible to convert any DDG into one which has no timestamp pair labels.

[**Theorem**] **(Completeness).** The set of optimizations OPT-1 through OPT-6 is *complete*.

[**Proof**] There are two types of edges in the DDG, data dependence and control dependence. Lets us consider each of the edge types and show that a list of labels associated with an edge can be eliminated using the optimizations described.

(1) *Data dependence labels.* (Local Edge) If the labels are associated with an edge that is local to a basic block the labels can be always removed because either they can be *inferred* and hence OPT-1a is applicable or they can be *entirely converted* to labels that can be inferred by carrying out specialization using OPT-2a. (Non-local Edge) If the labels are associated with an edge that is non-local, i.e. it connects two different basic blocks, then it can always be *converted into a local edge* by applying path specialization using OPT-2c. Once it has been converted to a local edge, its labels can always be eliminated as described above. Thus, it is concluded that labels associated with all data dependence edges can be eliminated by using the optimizations provided.

(2) *Control dependence labels.* (Fixed Distance from Unique Ancestor) If a node is

at a fixed distance from its control ancestor, then the labels can be *inferred* and hence optimization OPT-4 is applicable. (Others) If the node has multiple control ancestors and/or it is at a varying distance from its control ancestors, then path specialization using optimizations OPT-5a and OPT-5b can always be applied to *convert* the labels into ones that can be inferred. Thus, it is concluded that labels associated with all control dependence edges can be eliminated using the optimizations provided.

From (1) and (2) it is concluded that the optimization set is *complete*. □

It is worth noting that in the above proof no reference was made to optimizations OPT-1b, OPT-2b, OPT-3, and OPT-6. These optimizations are not needed for completeness. They are provided as cheaper alternatives to specialization in situation where they may be found to be applicable.

3.3 DDG Construction and Dynamic Slicing

In this section, the construction of a DDG and how to perform dynamic slicing on the DDG are discussed. First, lets discuss how a DDG is generated. In light of the previous discussions, this procedure consists of two steps: in the first step a static graph is constructed, which is transformed from a plain control flow graph as discussed in the WET representation, and in the second step dynamic information is labeled on the static edges.

Static Component of DDG. To construct the static component of DDG it is needed to perform the following analyses: (i) *reaching definitions* analysis is carried out to compute *def-use* information. *May alias* information is needed to carry out this analysis; (ii) *reaching uses* analysis is carried out to compute *use-use* information; (iii) *simultaneous reachability* analysis is carried out to identify situations in which a pair of non-local data dependence edges can *share labels*; and (iv) *postdominator*

analysis is carried out to compute *control dependences* [28]; and (v) *must reachability* analysis is carried out to identify situations in which a pair of non-local data and control dependence edges can *share labels*.

Except for *simultaneous reachability* analysis all other analyses are standard. Therefore next the details of the simultaneous reachability analysis will be described. Given a pair of definitions d_1 and d_2 in block s, with corresponding uses u_1 and u_2 in block d, the edges $d_1 \rightarrow u_1$ and $d_2 \rightarrow u_2$ will share identical labels if and only if whenever s and then d are executed either both data dependences are exercised or neither of them are exercised. The subgraph consisting of s, d, and all nodes along paths from s to d is considered. The set of nodes in this subgraph excluding s is referred to as $reach(sd)$. $KILL_n$ is a two bit value where bits correspond to the two definitions; bit value of 1 indicates that n does not kill the definition while 0 indicates that n kills the definition. The following equations compute for each node in $reach(sd)$ a data flow value which is $\subseteq \{11, 10, 01, 00\}$.

$$\forall n \in succ(s) \bigcap reach(sd), \quad x_n = \{11\}$$

$$\forall n \in reach(sd) - succ(s),$$

$$x_n = \bigcup_{p \in pred(n) \cap reach(sd)} \{KILL_p \wedge x : x \in x_p\}$$

If the solution for node d is $\{11\}$ (i.e., both definitions always reach d) or $\{11, 00\}$ (either both definitions reach d or neither reaches d), then the two dependence edges will always have identical labels. On the other hand, if the solution contains 10 (01), then there is a path from s to d along which d_1 (d_2) reaches d but d_2 (d_1) does not reach d. This analysis does not need to be carried out for every pair of definitions but rather for those that appear in the same basic block and have corresponding uses in the same basic block. Moreover, transitivity can be used to further reduce the pairs considered (i.e, if $(d_1 \rightarrow u_1, d_2 \rightarrow u_2)$ can share labels and $(d_2 \rightarrow u_2, d_3 \rightarrow u_3)$ can

share labels, then so can $(d_1 \rightarrow u_1, d_3 \rightarrow u_3)$).

Given the results of the above analyses, enough information is available to construct the static component of DDG. However, it is observed that the static component of DDG must be constructed once and then used repeatedly to capture dynamic dependence histories of different program runs. In other words the optimizations must be applied to construct the static component. While many of the optimizations can be applied for every opportunity that exists, there is a subset of optimizations that must be applied selectively. In particular, all of the specialization based optimizations should be applied only if it is expected that their application will result in more compaction than the graph expansion that is caused by specialization. Therefore these optimizations should be applied in a profile guided fashion. All Ball Larus paths [9] having a non-zero frequency during a profiling run were specialized. This approach works well because nearly all of the optimizations requiring specialization, are actually based upon path specialization. There are two optimizations that require data dependence profiles – OPT-1b and OPT-2a. The implementation does not make use of data dependence profiles yet. Instead OPT-1b was applied such that data dependence edges created due to must aliases were given priority for partial elimination over edges due to may aliases. OPT-2a is not applied because an effective static heuristic is not available to do so.

Dynamic Component of DDG. As the program executes, it sends a trace of one basic block at a time to an online algorithm which builds the DDG. This online algorithm must carry out two tasks. First it must buffer the basic block traces until it is determined which node in the static DDG must be augmented with additional dynamic edges. This is necessary because there may be multiple copies of a basic block due to specialization. Second it maintains the timestamp value and uses it to create the labels corresponding to the dynamic edges.

Consider the example shown in Figure 3.11. Let us assume that for the CFG

shown the static graph constructed has nodes for each of the basic blocks and another node for path 1245 is created due to specialization. When the program executes and generates a trace for block 1, at this time dependence edges can not be introduced for statements in 1 because it is not known where to introduce these edges – in copy of statements of 1 in node 1 or node 1245. The trace must be buffered till it is clear that either the program has followed path 1245 or that it has taken some other path. To detect when it is the right time to introduce edges the tree can be constructed as shown in Figure. 3.11(c). The online algorithm is initially at the root of the tree. Depending upon the basic block executed, the appropriate edge labeled with that block is traversed and the trace is buffered. When a leaf is reached, it is time to process the buffered trace. The leaf is labeled with the list of nodes in the DDG from which the edges introduced will originate. For example if basic blocks 1, 2, 4, and 5 are executed the edges originate from node 1245 while if blocks 1, 2, 4, and 6 are executed the edges originate from nodes 1, 2, 4, and 6.

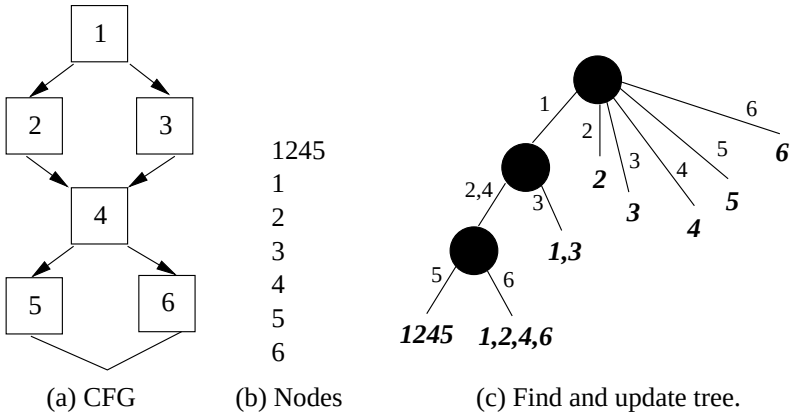


FIGURE 3.11. Introducing dynamic edges.

Dynamic Slicing. During the computation of a dynamic slice, the DDG is traversed backwards to identify the statements that belong to the slice. The set of dependence edges E_s going backwards from a statement s can be partitioned into subsets of edges E_{u_s} corresponding to each use u_s and subset of edges E_{c_s} corre-

sponding to all control ancestors of s. In other words, $E_s = \bigcup_{\forall u_s} E_{u_s} \bigcup E_{c_s}$. Given an execution instance of s, say $s < t_s >$, for each subset of edges corresponding to a dependence in E_s (i.e., E_{u_s} or E_{c_s}), it is needed to locate the specific edge $s's$ in E_s that must be followed. Moreover, since the edge $s' \rightarrow s$ may have been exercised many times, the specific dynamic instance of this edge $(s' \rightarrow s) < t_{s'}, t_s >$ that is involved in the dependence must be identified.

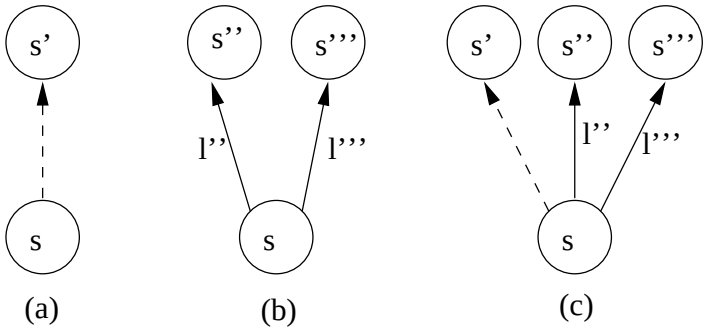


FIGURE 3.12. Traversing dependence edges.

There are three situations that arise as shown in Figure 3.12. Let us say a subset of edges $E_{d(s)}$ from E_s are being considered due to a dependence d involving s (i.e., $E_{d(s)}$ corresponds to some E_{u_s} or E_{c_s}). In the first case, $E_{d(s)}$ contains a single static edge $s' \rightarrow s$ which is thus not labeled dynamically with timestamp pairs. The traversal is straightforward as there is only one choice and the timestamp $t_{s'}$ in $(s' \rightarrow s) < t_{s'}, t_s >$ can be easily determined ($t_{s'} = t_s$ for data dependences and $t_{s'} = t_s - \delta$ for control dependences). In the second case there are multiple dynamic and thus labeled edges (say $s'' \rightarrow s$ and $s''' \rightarrow s$). The labels on these edges (l'' and l''') must be searched to locate the relevant edge and its instance – $(s'' \rightarrow s) < t_{s''}, t_s >$ or $(s''' \rightarrow s) < t_{s'''}, t_s >$. In the third case, there is a single unlabeled static edge $s' \rightarrow s$ as well as multiple labeled dynamic edges (say $s'' \rightarrow s$ and $s''' \rightarrow s$). The labels on $s'' \rightarrow s$ and $s''' \rightarrow s$ (i.e., l'' and l''') are first searched. If the relevant dependence $(s'' \rightarrow s) < t_{s''}, t_s >$ or $(s''' \rightarrow s) < t_{s'''}, t_s >$ is found, it is done; otherwise the static edge $s' \rightarrow s$ is selected and the value of timestamp $t_{s'}$ in $(s' \rightarrow s) < t_{s'}, t_s >$ is computed as discussed in the

first case.

It is worth noting that removal of explicit timestamps, as is carried out by the series of optimizations developed, not only makes the dependence graph more compact, it also speeds up the traversal process as fewer timestamps are searched to locate the relevant timestamp. The first and third cases contain a static unlabeled edge and hence the search is reduced while the second case represents the situation in which no reduction in search is achieved as all dynamic labels are saved and hence potentially searched.

The key points of the traversal process have been described. Now the dynamic slicing algorithm is summarized. In order to enable computation of slices, for each variable v the triple is maintained $< s, n, t_s >$ such that v was last defined by statement s in node n at time t_s. The dynamic slice for v is computed as shown below. Notice the manipulation of timestamps for unlabeled edges and also note that if $s' \rightarrow s$ is a uu-edge then s' is not added to the slice. The sharing of labels between different edges is not explicitly reflected in the algorithm below since it is an implementation detail which affects how the timestamp labels on edges are accessed. In the algorithm below, $sSlice(s < t_s >)$ represents the set of statements that belong to the dynamic slice of execution instance $s < t_s >$ and $eSlice(E, t_s)$ represents the subset of statements in the dynamic slice of execution instance $s < t_s >$ that are contributed by the traversal of the subset of dynamic edges E from $s < t_s >$.

$$BwS(v) = \{s\} \bigcup sSlice(s < t_s >)$$

$$sSlice(s < t_s >) = \begin{cases} if\ E_s = \bigcup\limits_{\forall u_s} E_{u_s} \bigcup E_{c_s} \neq \phi\ then \\ \qquad \bigcup\limits_{\forall u_s} eSlice(E_{u_s}, t_s) \bigcup eSlice(E_{c_s}, t_s) \\ else \\ \qquad \phi \\ endif \end{cases}$$

$$eSlice(E, t_s) = \begin{cases} if\ \exists\ labeled\ edge\ (s' \rightarrow s) < t_{s'}, t_s > \in E\ then \\ \quad sSlice(s' < t'_s >)\ \bigcup\{s'\} \\ elseif\ \exists\ unlabeled\ edge\ s' \rightarrow s \in E\ then \\ \quad case\ s's\ is: \\ \quad du\ edge:\ sSlice(s' < t_s >)\ \bigcup\{s'\} \\ \quad uu\ edge:\ sSlice(s' < t_s >) \\ \quad cd\ edge:\ sSlice(s' < t_s - \delta_{s' \rightarrow s} >)\ \bigcup\{s'\} \\ endif \end{cases}$$

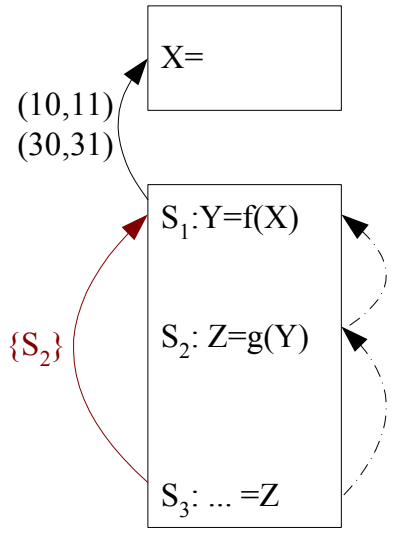


FIGURE 3.13. Using shortcuts.

Using Shortcuts to Speed Up Traversal. Finally, an optimization is presented that is used to speed up the traversal of the DDG by the above slicing algorithm. As shown, the optimized algorithm introduces some dependence edges statically while others are introduced dynamically. It is possible that at some points in the DDG multiple edges are traversed in sequence that are all static edges. If this is the case, the contributions to the dynamic slice when these edges are traversed is also known statically and always the same. Therefore, to speed up traversal of these edges, a *shortcut* edge is introduced that replaces the traversal of multiple static dependence edges by the traversal of a single shortcut edge. The edge is labeled with the set of statements that are skipped by the shortcut edge so that the dynamic slice can be appropriately updated when the shortcut edge is traversed. In the example shown

in Figure 3.13, corresponding to the sequence of two static edges $S_3 \rightarrow S_2 \rightarrow S_1$, a shortcut edge $S_3 \rightarrow S_1$ labeled with $\{S_2\}$ is introduced.

3.4 Experimental Results

The algorithm described has been implemented using the *Trimaran* compiler infrastructure which handles programs written in `C`. The experiments were performed on the same set of SPECInt benchmarks. The goal of the experiments was to essentially determine the space and time costs of the proposed dynamic slicing algorithm which is also referred to as OPT. It is also compared with the demand driven algorithm discussed in chapter 2.

3.4.1 Performance Evaluation of OPT

Graph sizes. The size of each full dynamic dependence graph is measured and it is compared with the size of the corresponding optimized graph obtained after application of all the optimizations described in this paper. These graph sizes are shown in Table 3.1. As shown in the table, the graphs sizes are reduced by factors ranging from 7.46 to 93.40. As a result, while the full graph sizes range in size from 0.8 to 1.95 Gigabytes, the optimized graphs range from 20 to 210 Megabytes in size.

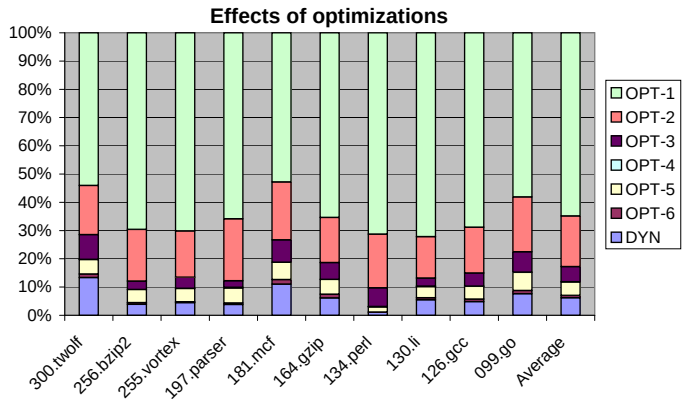


FIGURE 3.14. Effect of various optimizations on DDG size.

TABLE 3.1. DDG size reduction.

Program	Exec. (Millions)	Graph Size (Megabytes)		Reduction Factor
		Before	After	
`300.twolf`	140	1568.44	210.21	7.46
`256.bzip2`	67	1296.14	50.48	25.68
`255.vortex`	108	1442.66	64.81	22.26
`197.parser`	123	1816.95	69.81	26.03
`181.mcf`	118	1535.84	170.29	9.02
`164.gzip`	71	834.74	51.57	16.19
`134.perl`	220	1954.40	20.92	93.40
`130.li`	123	1745.72	96.50	18.09
`126.gcc`	131	1534.37	74.71	20.54
`099.go`	138	1707.36	131.24	13.01

The substantial reduction in the graph size is due to the fact that roughly only 6% of the dynamic dependences are explicitly maintained after the proposed optimizations are applied. The contributions of the various optimizations in reducing the graph size are shown in Figure 3.14. Here 100% corresponds to the full graph size and `dyn` corresponds to the size of the graph after application of all the optimizations. The other points in the bar graph show how the size reduces as the optimizations are applied one by one. As illustrated, OPT-1 is very effective as it reduces graph sizes to roughly 35% of the full graph size. However, the other optimizations also contribute significantly as they together reduce the graph size from 35% to 6% of the full graph size. It is important to point out that the distribution obtained is dependent upon the order in which the optimizations are applied since some cases can be handled by multiple optimizations.

It is observed that the majority of the savings comes from applying optimizations for dynamic data dependence edges. This is because the dynamic control dependences represent a small fraction of information contained in DDG (see the first graph in Figure 3.15). This is not surprising because control dependence edges are introduced at basic block granularity while data dependence edges have to be introduced at

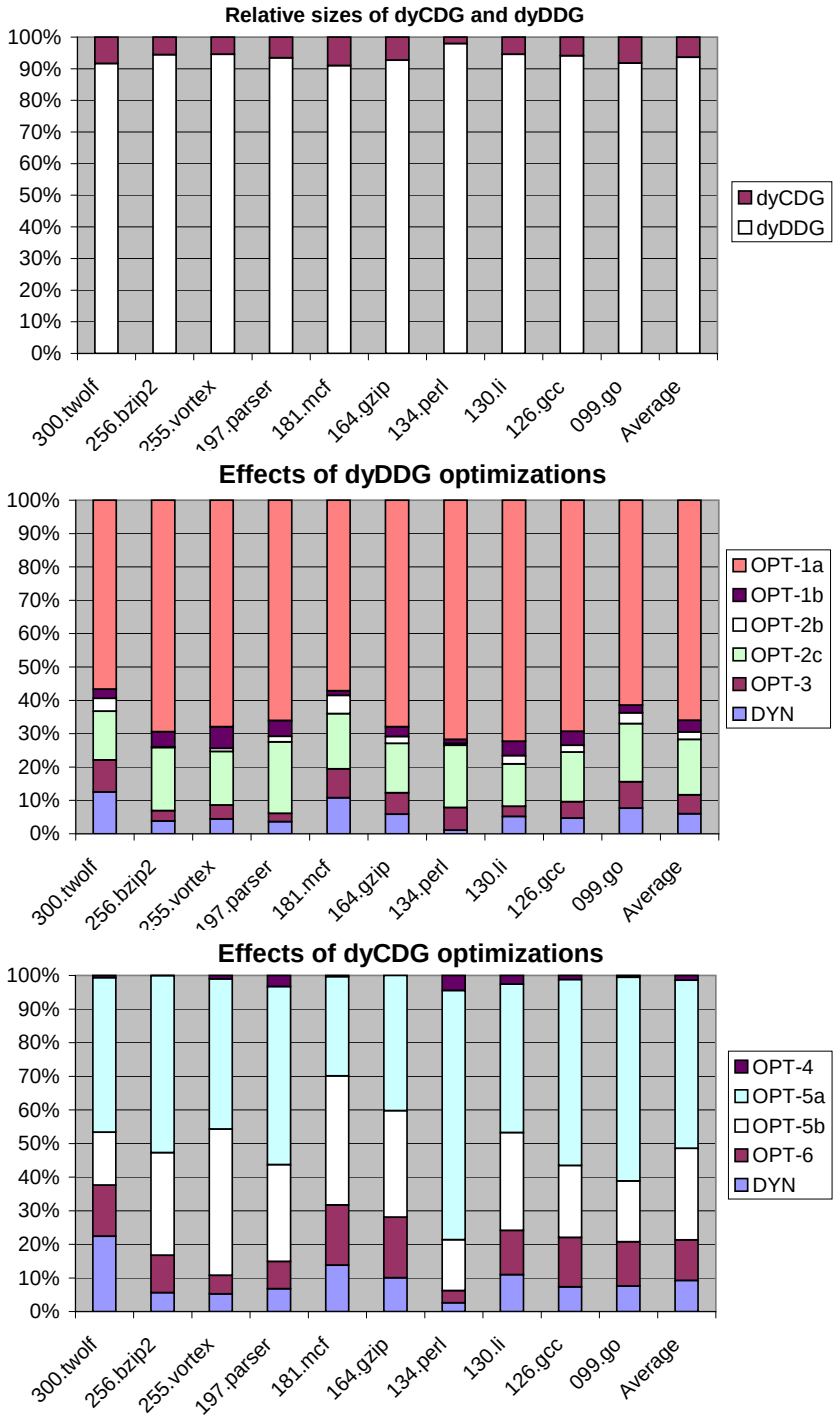


FIGURE 3.15. dyDDG vs. dyCDG size reduction.

statement granularity. The second and third graphs in Figure 3.15 separately show the reductions in the sizes of dyDDG and dyCDG due to the application of optimizations. The contributions of the individual optimizations is further breakdowned. Note that the second graph in Figure 3.15 does not include OPT-2a because it was never applied.

Execution times. The next step is to analyze the slicing times of the proposed algorithm. To carry out this study multiple program slices were computed at various points during the execution of each program. The reason why multiple slices were computed is that depending upon the memory address or variable chosen, the slicing times can vary. The reason why slicing was carried out at different points during execution is that the change in slicing times was wanted as the size of the DDG grows. Moreover this scenario also represents a realistic use of slicing – the user may want to compute slices at different execution points.

The results of this study are presented in Figure 3.16. In this graph each point corresponds to average slicing time for 25 slices. For each benchmark 25 new slices were computed after execution interval of 15 million statements – these slices corresponded to 25 distinct memory references. Following each execution interval slices were computed for memory addresses that had been defined since the last execution interval – this was done to avoid repeated computation of same slices during the experiment. As shown in the figure, the increase in slicing times is linear with respect to number of statements executed. More importantly the slicing times are very promising. For 8 out of 10 benchmarks the average slicing time for 25 slices computed at the end of the run is below 18 seconds. The only exception is `300.twolf` for which average slicing time at the end of the program run is roughly 36 seconds. It is worth noting that the optimizations did not reduce the graph size for this program as much as many of the other benchmarks. Finally, at earlier points during program runs the slicing times are even lower.

The above experiment was also performed without making use of the shortcut

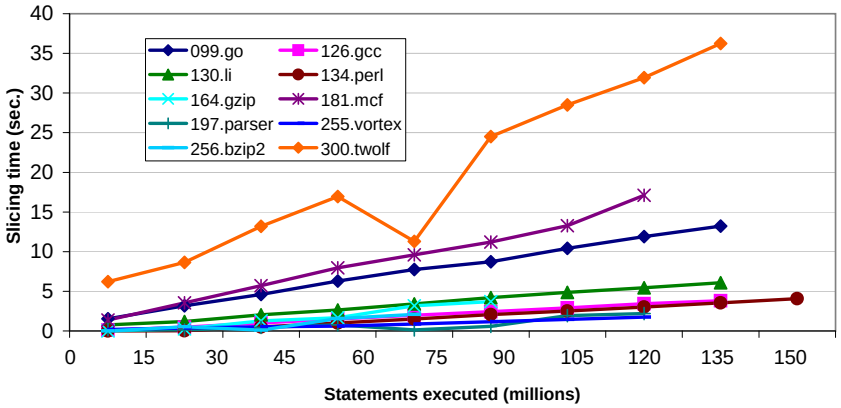


FIGURE 3.16. Dynamic slicing times of OPT.

edges in the DDG. The average slicing times at the end of the program run with and without making use of shortcuts are given in Table 3.2. In 8 out of 10 benchmarks, by making use of shortcuts, the average slicing time is cut by more than half. Thus, this is an important optimization.

Finally, let us consider the cost of generating the DDGs so that dynamic slicing can be performed. The implementation performs DDG construction in two steps. First instrumented programs are run to collect *execution traces* (control flow and data address traces). In the Trimaran environment, the execution of the program slows down roughly by a factor of two when traces are generated. Second execution traces are *preprocessed* to generate DDGs. The preprocessing times are shown in Table 3.3.

3.4.2 Comparison with Other Algorithms

The performance of OPT is also compared with the *demand driven* (DD) algorithm and the traditional dynamic slicing algorithm based on unoptimized DDGs (BAS). The DD algorithm was found to be the best overall in [86] as it does not run out of memory for reasonably long program runs. The traditional BAS algorithm runs out of memory for long program runs. However, in order to be able to successfully run

TABLE 3.2. Benefit of providing shortcuts.

Program	OPT slicing Times (seconds)		Ratio
	w/o shortcuts	with shortcuts	w/o / with
300.twolf	68.01	36.25	1.88
256.bzip2	6.14	2.10	2.92
255.vortex	5.57	1.92	2.90
197.parser	4.86	2.21	2.20
181.mcf	22.05	17.10	1.29
164.gzip	4.54	1.74	2.61
134.perl	12.59	4.05	3.11
130.li	15.65	6.09	2.57
126.gcc	9.76	3.80	2.57
099.go	26.85	11.36	2.36

TABLE 3.3. Preprocessing time for OPT.

Program	Preprocessing Time (Minutes)	Program	Preprocessing Time (Minutes)
300.twolf	65.29	256.bzip2	38.36
255.vortex	44.46	197.parser	44.06
181.mcf	53.64	164.gzip	23.52
134.perl	51.12	130.li	49.88
126.gcc	48.83	099.go	35.24

BAS, a machine with 2 Gigabyte RAM was used which was sufficient to accommodate the original DDGs for all but one program run (`134.perl`).

The cumulative slicing times for computing up to 25 slices at the end of the program run for the two algorithms are plotted in Figure 3.17. As one can see, the DD algorithm is much slower than the proposed algorithm. Computing each new slice using DD on an average takes 4.69 to 25.21 minutes depending upon the benchmark while computing the same slice using the optimized algorithm OPT takes 1.74 to 36.35 seconds. The DD algorithm spends a great deal of time traversing the execution trace stored on disk during each slice computation. The point at which the curves intersect the y-axis represents the preprocessing time – for the proposed algorithm this is the time for building the DDG while for the DD algorithm this is the time for preprocessing the execution trace to enable faster traversal of the trace as described in [86]. The exact preprocessing times are given in Table 3.4. As one can see, while the preprocessing time of the proposed OPT algorithm is higher, the difference is comparable to the time spent on computing a few slices using the DD algorithm.

The memory needed by the OPT and DD algorithms is given in Table 3.5. While the memory used by the OPT algorithm is the size of reduced DDG, the memory used by DD is the size of the DDG subgraph corresponding to a slicing request. Since the latter varies with slicing requests, the largest DDG subgraph size constructed in response to 25 distinct slicing requests is presented. Note that in 5 out of 10 benchmarks the size of the largest DDG subgraph built by DD is greater than the full reduced DDG built by OPT. It is clear from this data that on average, the memory needs of DD and OPT are comparable to each other.

Therefore, based upon the above results it can be concluded that OPT is superior to DD because it is much faster than DD and at the same time it uses roughly the same amount of memory as DD.

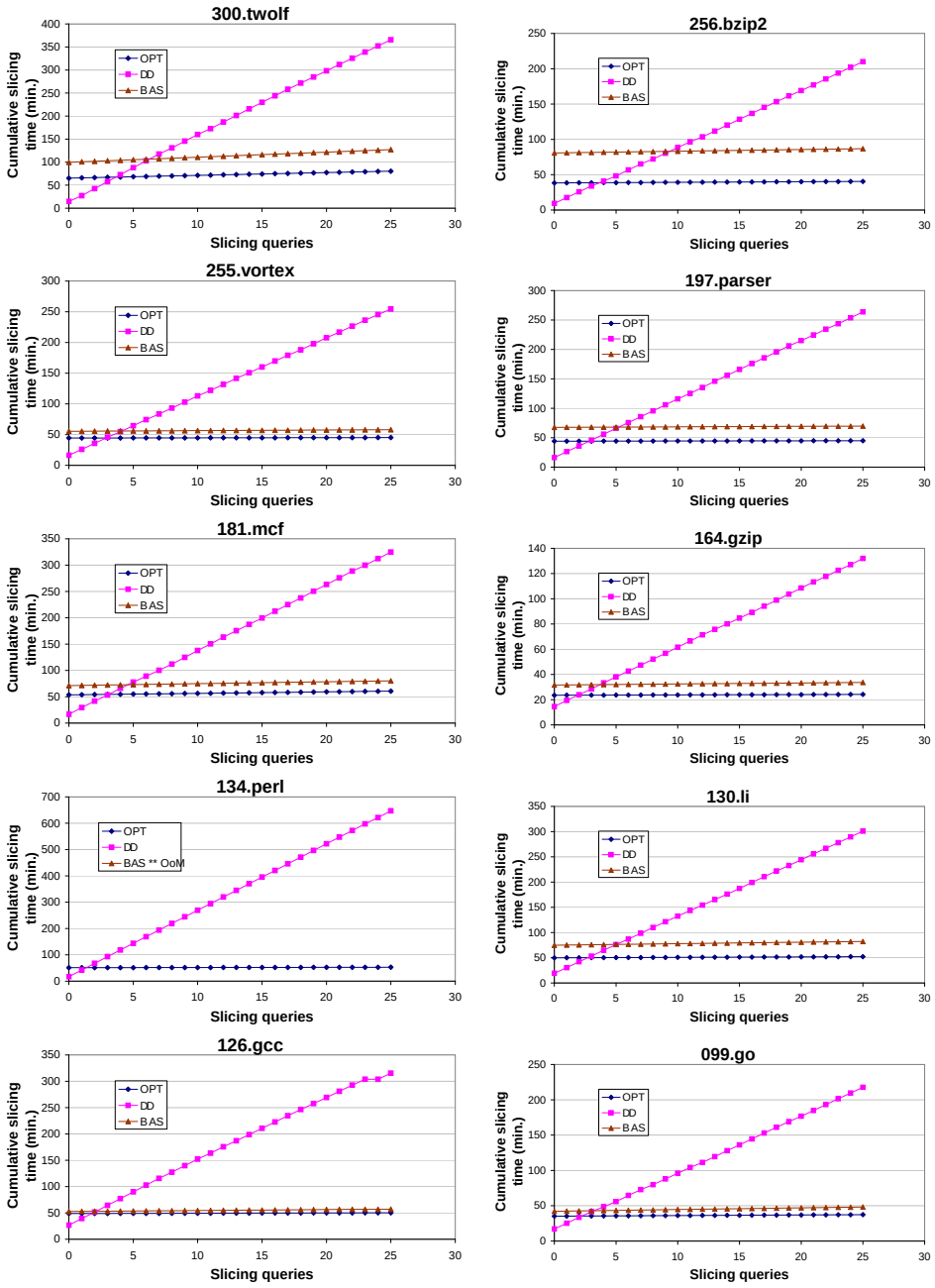


FIGURE 3.17. Comparison of OPT with DD and BAS.

BAS versus OPT. As mentioned earlier, BAS runs out of memory for reasonably long program runs. Fortunately, the slicing times of the OPT algorithm can be compared to that of the BAS algorithm in situations where the program run was short enough to enable the entire (unoptimized) dynamic dependence graph to be kept in memory. BAS was able to successfully run on a machine with 2 Gigabyte RAM for all programs except `134.perl`.

The slicing times of BAS and OPT are compared in Table 3.6. It is observed that OPT is faster than BAS. This is because of the use of shortcut edges that speed up the traversal of the DDG. The same optimization cannot be applied to BAS because the unoptimized DDG only contains dynamic edges. The difference between OPT slicing times and BAS slicing times are caused by shortcuts because when the slicing times of OPT without shortcuts (given earlier in Table 3.2) is compared with slicing time of BAS given below, they are quite close.

Finally the preprocessing times of OPT and BAS are compared. It had been expected that the preprocessing times for OPT would be higher than BAS because the OPT algorithm must spend some extra time for checking whether the timestamp pairs of all exercised dependences should be added to the DDG or not. However, the experiments show otherwise. As the data in Table 3.7 shows, the preprocessing times for the BAS algorithm are consistently higher than those for OPT. The reason for this behavior is that the list of timestamp pairs that are associated with a dependence edge often grow very large and thus resizing of the array in which these are stored must often be performed. These memory reallocation operations take up significant amount of time in BAS while in OPT this is not the case. Thus, the overall effect of this behavior is that the preprocessing times of OPT are lower than that of BAS.

Therefore, based upon the above results one can say that OPT is superior to BAS not only because it scales to longer program runs, but also because it has lower preprocessing and slicing times.

TABLE 3.4. Preprocessing time: DD vs. OPT.

Program	Preprocessing Time (Minutes)		Ratio
	OPT	DD	DD/OPT
300.twolf	65.29	14.54	0.22
256.bzip2	38.36	9.38	0.25
255.vortex	44.46	16.35	0.37
197.parser	44.06	16.23	0.37
181.mcf	53.64	16.64	0.31
164.gzip	23.52	14.56	0.62
134.perl	51.12	17.18	0.34
130.li	49.88	19.23	0.39
126.gcc	48.83	26.65	0.55
099.go	35.24	17.06	0.48

TABLE 3.5. DDG graph sizes: DD vs. OPT.

Program	Graph Size (Megabytes)	
	OPT	DD (Max. of 25 slices)
300.twolf	210.21	296.06
256.bzip2	50.48	80.66
255.vortex	64.81	33.60
197.parser	69.81	40.04
181.mcf	170.29	113.74
164.gzip	51.57	34.75
134.perl	20.92	53.62
130.li	96.50	105.45
126.gcc	74.71	57.70
099.go	131.24	162.28

TABLE 3.6. Slicing times: BAS vs. OPT.

Program	Slicing Times (seconds)	
	BAS	OPT
300.twolf	65.99	36.25
256.bzip2	5.92	2.10
255.vortex	6.17	1.92
197.parser	5.28	2.21
181.mcf	21.71	17.10
164.gzip	4.83	1.74
134.perl	Out of Mem.	4.05
130.li	17.86	6.09
126.gcc	11.03	3.80
099.go	29.79	11.36

TABLE 3.7. Preprocessing time: BAS vs. OPT.

Program	Preprocessing Time (Minutes)		Ratio
	OPT	BAS	BAS/OPT
300.twolf	65.29	99.62	1.53
256.bzip2	38.36	80.78	2.11
255.vortex	44.46	55.47	1.25
197.parser	44.06	67.57	1.53
181.mcf	53.64	71.17	1.33
164.gzip	23.52	31.66	1.35
134.perl	51.12	Out of	Mem.
130.li	49.88	74.86	1.50
126.gcc	48.83	52.70	1.08
099.go	35.24	42.17	1.20

3.5 Summary

In conclusion, the OPT algorithm proposed in this chapter provides fast slicing times (1.74 to 36.25 seconds) and compact dynamic dependence graph representation (20 to 210 Megabytes) leading to a space and time efficient algorithm for dynamic slicing. In contrast, the prior algorithms are either space inefficient (corresponding graph sizes for FP are 0.84 to 1.95 Gigabytes) or time inefficient (corresponding slicing times for LP are 4.69 to 25.21 minutes) making them unattractive for use in practice. The development of a cost effective dynamic slicing algorithm is an important contribution as a wide range of applications that require analysis of dynamic information are making use of dynamic slicing. Next chapter discusses how different types dynamic slices are computed based on dependence profile to produce a small fault candidate set.

Chapter 4
Effectiveness of Dynamic Slicing

To construct a fault candidate set, a slicing criterion needs to be identified, which is a value in the dynamic dependence graph that is related to the execution of the faulty code. Once the slicing criterion is known, the dynamic dependence graph is traversed to identify the set of statements that are related to this value through chains of dynamic dependences. This set of statements includes the faulty code as well and is therefore a possible fault candidate set. So far the slicing criterion that has been introduced is the incorrect output, from which a backward dynamic slice is computed to provide a fault candidate set. In this chapter, two new types of slicing criteria and the corresponding dynamic slices are introduced. They are further used in combination with the backward dynamic slice to produce smaller fault candidate sets.

- The first new type of dynamic slice is based upon a *minimal failure inducing input difference*. Given an input on which the program fails, the *minimal failure inducing input difference* is the part of the input that is found to cause the failure. As a result the *forward dynamic slice* (FwS) starting from the failure inducing input produces yet another fault candidate set that captures the faulty code.

- The second new type of dynamic slice is based upon *critical predicates*. Given an input on which the program fails, a *critical predicate* is an execution instance of a conditional branch predicate such that if the outcome of the predicate's execution instance is reversed, the program terminates producing the correct output. Since the predicate outcome is related to the fault, it is found that

the *bidirectional dynamic slice* (BiS) which includes both the backward and forward dynamic slices of the predicate execution instance produces another fault candidate set.

The three types of dynamic slices (backward, forward, and bidirectional slices) are also called *single point dynamic slices* because the construction of the dynamic slice begins starting from a value that is known to be related to the cause of program failure (i.e., the faulty code executed in the failed run). Finally, now that there exist three different fault candidate sets corresponding to the backward, forward, and bidirectional slices, an even smaller fault candidate set can be produced by intersecting two or more fault candidate sets corresponding to the three different types of slices. Such resulting slices are referred to as *multiple points dynamic slices* (MPS) as they are the result of computing dynamic slices starting from multiple points (erroneous value, failure inducing input, and critical predicate).

4.1 Forward Dynamic Slice of Minimal Failure-Inducing Input Difference

Let us first begin by briefly discussing the concept of *minimal failure inducing input difference*, which is the slicing criterion for computing a *forward slice*. Zeller introduced the term of delta debugging [80] for the process of determining the causes for program behaviors by looking at the differences (the deltas) between the old and new configurations of the programs. Hildebrandt and Zeller [41, 83] then applied the delta debugging approach to simplify and isolate the failure inducing input difference. The basic idea of delta debugging is as follows. Given two program runs r_s and r_f corresponding to the inputs I_s and I_f respectively, such that the program fails in run r_f and completes execution successfully in run r_s, the delta debugging algorithm can be used to systematically produce a pair of inputs I'_s and I'_f with minimal difference such that program fails for I'_f and executes successfully for I'_s. The difference be-

tween these two inputs isolates the minimal failure inducing difference part of the input. These inputs are such that their values play a critical role in distinguishing a successful run from a failing run.

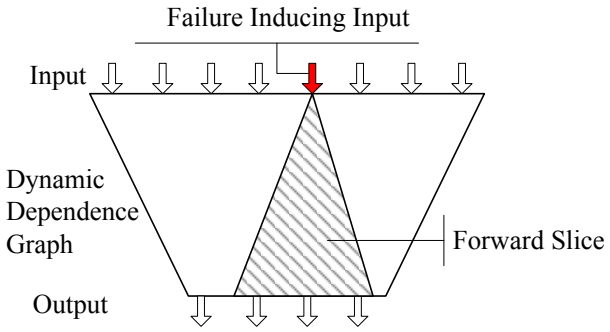


FIGURE 4.1. Forward dynamic slice.

Since the minimal failure inducing input difference leads to the execution of faulty code and hence causes the program to fail, a fault candidate set can be identified by computing a *forward dynamic slice* starting at this input. In other words all statements that are influenced by the input value directly or indirectly through a chain of data and/or control dependences are included in the fault candidate set. Thus, forward dynamic slicing was recognized as a means of producing a new type of dynamic slice which also represents a fault candidate set for the first time [31].

The outline of the algorithm is given in Algorithm 1.

Algorithm 1 *Fault location using minimal failure inducing input difference.*

Step 1: Compute minimal failure inducing input by:
    either use *ddmin* to Simplify input [83]:
        $I'_f = ddmin(I_f)$
        $\Delta_{min} = I'_f$
    or use *dd* to Isolate input difference [83]:
        $(I'_s, I'_f) = dd(I_s, I_f)$
        $\Delta_{min} = I'_s$ - I'_f
Step 2: Compute forward dynamic slice:
    $FwS = FwdSlice(I'_f, \Delta_{min})$

Step 1: Finding minimal failure inducing input. To find a failure inducing input, any of the two algorithms given by Zeller and Hildebrandt in [83] can be used. The first algorithm (ddmin) *simplifies* a failing test case I_f to produce a minimal test case I'_f such that removing any single input entity from I'_f causes the failure to disappear. Therefore Δ_{min} is I'_f in this case. The second algorithm (dd) *isolates* a minimal failure inducing input difference between a failing and a passing test case. Given inputs I_f and I_s for a failed run and a successful run respectively, this algorithm returns a pair of inputs (I'_f, I'_s), such that I'_s and I'_f correspond to a successful run and a failed run respectively and any single part of $I'_f - I'_s$ if removed from I'_f would make the failure disappear or if added to I'_s would make the failure occur. Therefore in this case Δ_{min}=I'_f-I'_s.

Step 2: Compute Forward Dynamic Slice. The minimal failure inducing input difference Δ_{min} computed by the first step defines the slicing criteria for the forward dynamic slicing. In this step, the forward dynamic slice $FwS = FwdSlice(I'_f, \Delta_{min})$ is computed. Given an input and the corresponding execution, the *forward dynamic slice* is the set of statements which are affected by that particular input via data/control dependences. The forward dynamic slicing algorithm $FwdSlice$ is similar to the backward dynamic slicing algorithm presented in chapter 3 except that the dependence edges are traversed in the forward direction.

There is an additional cost to using the above technique. First the user must provide a parser for the input such that the input can be separated into meaningful entities and hence new inputs can be generated from them. Second the user must examine the outputs for the additional program runs corresponding to the generated inputs.

The above approach was applied to the buffer overflow error in $gzip - 1.0.7$ as described in Figure 4.2. Figure 4.2 illustrates the details of the problem. On the left hand side of Figure 4.2, the relevant code segment is shown. The problem happens

in the *strcpy* statement at line 844. Variable *ifname* is a global array defined at line 198. The size of the array is defined as 1024. Before the *strcpy* statement at line 844, there is no check on the length of string *iname*. If the length of string *iname* is longer than 1024, the buffer overflows and the program crashes. The memory

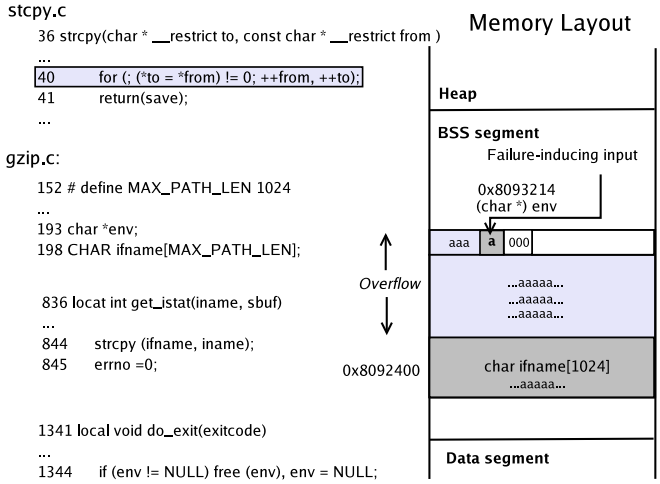


FIGURE 4.2. Buffer overflow bug in *gzip*.

layout of the *gzip* program is shown on the right side of Figure 4.2. As shown, there is a global pointer *env* located in an address space above array *ifname*. The difference between *env* and *ifname* is 3604 bytes. If the length of string *iname* is larger than 3604, the value of *env* will be changed due to buffer overflow. At line 1341 of function *do_exit*, before the program quits, it tries to free the memory pointed to by *env*. If the value of *env* is an illegal memory address due to buffer overflow, it causes a segmentation fault at line 1344. Two inputs were picked: the first input is a file name $'aaaaa'$, which is a successful input, and the second input is a file name $'a$ *< repeated 3610 times >*$'$, which is a failure input because the length is larger than 3604. After applying *sddmin* algorithm [83] on them, two new inputs resulted: the new successful input was a file name $'a$ *< repeated 3604 times >*$'$ and the new failed input was a file name $'a$ *< repeated 3605 times >*$'$. The failure inducing input difference between them was the last character 'a' in the new failed input.

Next, the forward slice was computed on the failure inducing input difference in the failed input. The size of the forward slice was 3 which includes the `for` statement at line 40 in *strcpy.c*. This is of course the place where the buffer overflow occurred. The slicing implementations run on the binary code level and thus are able to check the memory space of a program and even check the code in the library.

4.2 Bidirectional Dynamic Slice of a Critical Predicate

Given an erroneous run of the program, the objective of this method is to explicitly force the control flow of the program along an alternate path at a critical branch predicate such that the program produces the correct output. The basic idea of this approach is inspired from the following observation. Given an input on which the execution of a program fails, a common approach to debugging is to run the program on this input again, interrupt the execution at certain points, make changes to the program state, and then inspect the impact of changes on continued execution. If the changes to program state that cause the program to terminate correctly can be automatically discovered, it becomes much easier to understand the error. However, automating the search of state changes is prohibitively expensive and difficult to realize because the search space of potential state changes is extremely large (e.g., even possible changes for the value of a single variable are enormous if the type of the variable is integer or float). On the other hand, changing the outcomes of predicate instances greatly reduces the state search space since a branch predicate has only two possible outcomes, true or false. Therefore note that through forced *switching* of the outcomes of some predicate instances at runtime, it may be possible to cause the program to produce correct results.

Having identified a critical predicate instance, a fault candidate set is computed as the *bidirectional dynamic slice* of the critical predicate instance. This bidirectional slice is essentially the union of the backward dynamic slice and the forward dynamic

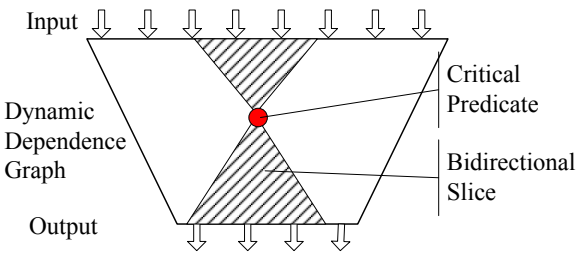


FIGURE 4.3. Bidirectional dynamic slice.

slice of the critical predicate instance. Intuitively, the reason why the slice must include both the backward and forward slice is as follows. Consider the situation in which the effect of executing faulty code is to cause the predicate to evaluate incorrectly. In this case the backward dynamic slice of the critical predicate instance will capture the faulty code. On the other hand it is possible that by changing the outcome of the critical predicate instance the execution of faulty code is avoided and hence the program terminates normally. In this case the forward slice of the critical predicate instance will capture the faulty code. Therefore, the faulty code will either be in the backward slice or the forward slice. The role of bidirectional dynamic slices in fault location was first recognized for the first time in [84].

The notion of critical predicate is illustrated with the faulty version of the *flex* (a fast lexical analyzer generator) program shown in Figure 4.4 which is taken from the Siemens suite [43, 2]. The Siemens suite provides the associated test suites and faulty version for each program. The program in Figure 4.4 is derived from $flex - 2.4.7$ and augmented by the provider of the program with a bug that is circled in the figure: $base[i + 1]$ should actually be $base[i]$. The first provided input which produced an erroneous output was taken. It was observed that the output is different from the expected output for the 538th character, a '1' is produced as output due to the execution of *printf* in the *else* part (line 2696) of the *else if* statement at line 2690 instead of a '0' that should be output by execution of the *printf* in the *then* part of the *else if* statement. Under the correct execution at line 2673 *offset* would have been

assigned the value of $base[0]$ which is 1. The variable $chk[0]$ at line 2681 would have

```
 970     base = ...
         . . .
2565     base[...] = ...
         . . .
2667     for ( i = 0; i <= lastdfa; ++i )
2668         {
             . . .
2673         int offset = base[i+1];
             . . .
2677         chk[offset] = EOB_POSITION;
             . . .
2681         chk[offset - 1] = ACTION_POSITION;
             . . .
2683         }
2684
2685     for ( i = 0; i <= tblend; ++i )
2686         {
             . . .
2690         else if ( chk[i] == ACTION_POSITION )
                 printf("%7d, %5d,", 0 , ...);
             . . .
2696         else    /* verify, transition */
                 printf("%7d, %5d," , chk[i], ...);
             . . .
2699         }
```

FIGURE 4.4. Incorrect output bug in *flex*.

been assigned $ACTION\_POSITION$ causing the predicate at line 2690 to evaluate to *true* for loop iteration corresponding to $i = 0$. Due to the error at line 2673, an incorrect value of $offset(= 3)$ causes $ch[0]$ to have an incorrect stale value $(= 1)$ which causes the predicate at line 2690 to incorrectly evaluate to its *false* outcome. The proposed method looked for a predicate instance whose switching corrected the output. The appropriate instance of the *else if* predicate instance was found through this search. Once this predicate instance was found, it could be easily determined by following backwards the data dependences that the incorrect value of $ch[0]$ was a stale value and it did not come from most recent execution of for loop at line 2667. Thus, now it was clear that the error was in the statement at line 2673 which set the *offset* value.

In the above example, enforcing the outcome of a predicate avoided the need to search for potential modifications of values for $chk[\,]$, *offset*, or $base[\,]$ which are *integer* variables and thus can take many different values. The above example also illustrates that it is important to alter the outcome of selected predicate instances

as opposed to all execution instances of a given predicate. This is because the fault need not be in the predicate but elsewhere and thus all evaluations of the predicate need not be incorrect.

The effectiveness of this approach can be explained as follows. Given a program run, from the perspective of a program output, the computation performed to compute an output can be divided into two parts: the *Data Part* (*DP*) and the *Select Part* (*SP*). The *Data Part*, *DP*, consists of executed instructions which compute data values that are involved in computing the actual value that is output. These instructions can be identified by computing the *backward data slice*, i.e. transitive closure of dynamic data dependences starting from the output value. The *Select Part*, *SP*, is the part of the computation that caused the selection of instructions in the dynamic data slice for execution. The presence of faulty code in *SP* may cause an incorrect dynamic data slice to be selected for execution and thus the generation of a wrong output value. In contrast to *DP*, the size of *SP* is much bigger. The study in [84] shows that the number of executed instructions in *DP* is 3 to 7 times smaller than the number of executed instructions in *SP* and the number of executed predicates is only a very small percentage of the total executed instructions in *SP*. This study suggests two points. First, if an error gets exercised, it is more likely the error occurs in the *SP* part of the execution. Second, since computation in *SP* eventually takes effect through predicates, switching the faulty predicate actually *fixes* the entire faulty computation in *SP* that contributes to the branch outcome of the faulty predicate. That is to say, switching predicates is a natural solution to locate predicate related faults.

4.2.1 Finding the Critical Predicate

In this section a detailed algorithm is developed for predicate switching. The general idea of the approach is to perform repeated executions of the program on the failing

input and switch conditional branch outcomes during these re-executions till a predicate switching is found that causes the program to produce the correct output. In doing so, it is the goal to develop a search strategy that is both practical and effective. To achieve this goal a search strategy is designed which incorporates the following features that together limit the search space and order the search.

Even though predicate switching greatly reduces the search space by limiting state changes to conditional branch outcomes, there are still a substantial number of instances of executed conditional branches whose outcomes are candidates for switching. Therefore, the search is limited so that during each new program execution, the outcome of only a *single predicate instance* is switched. In other words, the program behavior of simultaneously switching outcomes of multiple predicate instances is not explored because the number of such possibilities is very large.

- *Last Executed First Switched (LEFS) Ordering.* It has been decided that the outcome of one branch predicate instance should be switched in each re-execution. Next, the order in which possible predicate switchings are explored is discussed. One simple ordering strategy that is employed is based upon the following observation [61]: *execution of faulty code (i.e., root cause of a failed run) is often not far away from the point at which the program fails (e.g., program crashes or it produces a wrong output value).* Therefore possible predicate switchings are explored in the reverse order of the predicate executions, i.e. the outcome of the conditional branch instance encountered last is switched first.

- *Prioritization-based (PRIOR) Ordering.* In addition to the simple *LEFS* ordering strategy, another more aggressive prioritization based ordering strategy (*PRIOR*) is further developed. This strategy consists of two main steps. In the first step an algorithm is used to partition the set of all branch predicate instances into two subsets: those that are expected to be influenced by the faulty code via dependences and those that are not expected to be influenced by faulty

TABLE 4.1. Search strategies: *LEFS* vs. *PRIOR*.

Program	Total Instr.	After Fault	Dep . dist.	Total Preds	LEFS	PRIOR
flex 2.5.319(a)	17,637	2,583	23	3,669	432	6
flex 2.5.319(b)	366,624	search		60,481	failed	
flex 2.5.319(c)	303,121	search		46,820	failed	
grep 2.5	21,001	416	27	2,555	61	56
grep 2.5.1 (a)	4,290	232	26	844	38	21
grep 2.5.1 (b)	10,337	search		1,652	failed	
grep 2.5.1 (c)	41,068	185	15	9,561	32	28
make 3.80 (a)	1,907,361	163,050	23	166,837	155,492	102
make 3.80 (b)	1,787,616	404,400	50	218,778	50,909	7,108
bc-1.06	68,336	15,676	6	9,684	2,079	2
tar-1.13.25	2,471	1,783	12	470	388	3
tidy	771,154	108	3	131,336	39	1
s-flex-v4	321,888	11,728	5	59,352	4,228	37
s-flex-v5	171,953	search failed		30,203	error in DP	
s-flex-v6	8,252	search failed		1,717	error in DP	
s-flex-v7	187,903	139,799	21	33,136	26,028	6
s-flex-v8	9,848	1,522	NA	1,943	218	NA
s-flex-v9	69,258	59,209	33	13,010	11,085	190
s-flex-v10	177,821	41	16	29,240	4	4
s-flex-v11	185,724	27,809	11	33,199	7,189	13

code. The ones in the first subset are explored before the ones in the second subset. In the second step the branch predicate instances are ordered within the first subset according to their dynamic dependence distance from the erroneous output value. More precisely, the predicate instances that are separated by fewer dependence edges from the erroneous output value are explored before those that are separated by greater number of dependence edges. The resulting ordering of branch predicate instances is then used in our search.

Next, data is presented confirming the observations on which the above design choices are made. This data is based upon the set of real bugs presented in Table 2.3 in chapter 2 and a set of injected bugs taken from the faulty versions of *flex* provided in the Siemens suite [43], denoted as *s-flex*. Now let us consider the data in Table 4.1.

The total number of instructions executed, excluding the instructions from library code, before program terminated during a failing run is given first in column (*Total Instns*). In 15 out of 20 faults that were studied a predicate instance was found, switching which caused the failure to be removed. In three cases (faults (b) and (c) in *flex 2.5.319*; fault (b) in *grep 2.5.1*) the searching technique could not identify the critical predicate because the error is too complex for any predicate switch to produce correct output value. In versions 5 and 6 of *s-flex*, the search failed because the errors were in the data part of the computation. The number of instructions executed by the program following the execution of the critical predicate and before the program's termination are given in column *After Fault*. This number is considerably smaller than the number in *Total Instrns*. This difference motivates the *LEFS* strategy. The next column, *Dep. Dist.*, is the shortest dependence distance between the output and the critical predicate in the dynamic dependence graph. As one can see, this distance is quite small and thus this provides the motivation for our *PRIOR* strategy. The remaining data in the table demonstrates the effectiveness of the two search strategies. The column labeled *Total Preds* is the total number of conditional branches that were executed during the program runs while the last two columns, *LEFS* and *PRIOR*, give the number of predicate instances that were actually explored by switching before finding a predicate instance whose switching produced the correct output (i.e., a critical predicate). As we can see, these numbers are considerably smaller than the numbers in *Total Preds*. In addition, the *PRIOR* number is far smaller than the *LEFS* number in most of the cases. In the case of *s-flex-v8*, although a critical predicate was found using the *LEFS* strategy, one could not be found using the *PRIOR* strategy. This is because the dynamic slices could not be computed on which the ordering of predicate instances is based. Overall, the above data indicates that the more aggressive *PRIOR* strategy for ordering predicate instances is very effective.

Given the choices in search strategies described above, the predicate switching algorithm is presented. The overview of the algorithm is given in Figure 4.5. The

algorithm has three major steps: finding the first erroneous value in a failing run; identifying the predicate instances which will be considered using predicate switching; and finally searching for a critical predicate reversal of whose outcome causes the program run to succeed. Let us consider the above steps in greater detail:

Step 1: Locate the first erroneous output value. A program run is considered to be a failing run if it produces incorrect output. Given the correct output, the first deviation between the output produced by the failing run and the correct output and also identify the execution instance I_e of instruction I that produced the erroneous output value. The goal of the algorithm is to find a predicate instance switch that causes correct output value to be produced.

Step 1: Find Erroneous Value

Examine failed run to identify the first erroneous value
– erroneous output or value that crashes the program.

Step 2: Find Predicates for Switching

(a) Run the program again, generate *Predicate Trace* (PT) identifying all instances of branch predicates executed and their execution order.
(b) Perform *Predicate Ordering* of predicates in PT using *LEFS* or *PRIOR*.

Step 3: Find Critical Predicate

for each pred. instance P in ordered PT do
    Generate instrumented program to switch P's outcome;
    Execute above program;
    if program run succeeds, report P and terminate search.
endfor

FIGURE 4.5. Algorithm overview.

As mentioned earlier, if the program crashes at some execution instance I_e of instruction I, the value or address referenced by I_e that caused the program to crash takes the place of the erroneous output value. The goal of the algorithm in this situation is to find a predicate instance switch such that, following the switch, when execution instance I_e is encountered, the program does not crash.

Step 2: Identify predicate instances for switching. In this step the program is re-

run and the *Predicate Trace* (PT) is collected. The predicate trace is a record of all instances of conditional branches executed during the failing program run from the start of the execution to the point at which the failing run produced the erroneous value identified in the preceding step (i.e., when I_e was executed). The program execution performed in this step not only generates the predicate trace, but in addition it also provides information using which predicate instance ordering is performed. If *LEFS* is used, the ordering is already clear from the predicate trace. If *PRIOR* is used the ordering is performed as follows. The *dynamic dependence graph* is generated containing both dynamic data and control dependences during this run. Partitioning of predicates into high and low priority subsets is performed using dynamic slicing. The predicate instances that belong to the computed dynamic slice form the subset that contains predicate instances that were involved in producing the erroneous output. The remaining predicate instances form the lower priority subset. The predicate instances in the higher priority subset are further arranged in the order of increasing dependence distance from the erroneous output. The distances needed to perform this ordering are obtained from the dynamic dependence graph.

Step 3: Searching for a critical predicate. Figure 4.6 pictorially shows the search for a critical predicate when the simple *LEFS* strategy is used. The first line represents the failed run up to the point it produced the first erroneous value. The small ovals mark execution of predicate instances which are also labeled. Then the subsequent steps show how the predicate instances are switched one at a time in each subsequent run in the *LEFS* order. The predicate instance that is switched is marked using a larger oval. During each run the new output value is observed. The above process is repeated till correct output value is generated. The basic functioning of this step is the same when *PRIOR* strategy is used, only the order in which predicate instances are switched changes.

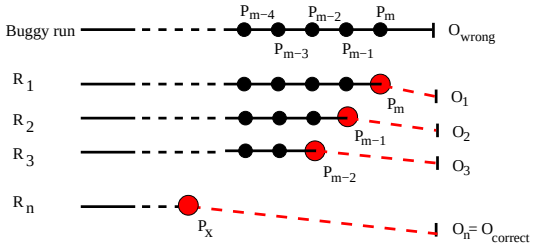


FIGURE 4.6. Search method.

4.2.2 Results of Searching for Critical Predicates

This section presents the evaluation of the critical predicate searching algorithm. Table 4.2 shows how often the algorithm is successful in finding a critical predicate. As column *Found* shows, in 15 out of 20 cases a predicate instance switch was found which caused the program to produce correct output or eliminated the cause of the program crash. The critical predicate identified is indicated in columns *Where* and *Which*. Here column *Where* gives the file name and source line number at which the switched predicate can be found and *Which* is the dynamic instance of the predicate that was switched. The predicate instance number is measured from the point at which erroneous output is produced or program crashed. A value of 0 corresponds to the most recent execution instance of the predicate while greater values correspond to earlier instances of the predicate. As one can see, in many cases the most recent instance of a predicate is the critical instance while in some cases it is not the most recent instance. Finally, column *False +ves* represents the number of dynamic predicate switches, which produced correct output but were not related to the fault, that were found by *PRIOR* (except in case of *s-flex-v8* which uses *LEFS*) before the desired predicate switch was located. As shown by the results, in all cases except one, this number is 0 indicating that the first predicate switch located by *PRIOR* was related to the fault. In one case first predicate switch found was not useful but the second one found was meaningful.

It has already been shown earlier that *PRIOR* locates the desired predicate instance switch far sooner than *LEFS*. Now the time taken by *PRIOR* to locate the

TABLE 4.2. Successful/Failed searches.

Program	Found	Where	Which	False +ves
flex 2.5.319(a)	yes	gen.c @ 1813	0	0
flex 2.5.319(b)	no	search failed		
flex 2.5.319(c)	no	search failed		
grep 2.5	yes	grep.c @ 532	0	0
grep 2.5.1 (a)	yes	search.c @ 549	0	0
grep 2.5.1 (b)	no	search failed		
grep 2.5.1 (c)	yes	dfa.c @ 2854	2	0
make 3.80 (a)	yes	read.c @ 6162	143	1
make 3.80 (b)	yes	remake.c @ 652	1	0
bc-1.06	yes	storage.c @ 176	9	0
tar-1.13.25	yes	prepargs.c @ 81	0	0
tidy	yes	parser.c @ 3496	0	0
s-flex-v4	yes	flex.c @ 2978	0	0
s-flex-v5	no	search failed – error in DP		
s-flex-v6	no	search failed – error in DP		
s-flex-v7	yes	flex.c @ 9171	0	0
s-flex-v8	yes	flex.c @ 11833	0	0
s-flex-v9	yes	flex.c @ 5046	0	0
s-flex-v10	yes	flex.c @ 2687	1	0
s-flex-v11	yes	flex.c @ 3559	0	0

desired predicate instance switch is presented. The results are given in Table 4.3. As one can see, the time taken to locate critical predicates is quite reasonable. In many cases it is around 1 minute. The cases in which the search failed, the time is large (few hours) as it went through all the predicate instances.

TABLE 4.3. Search time.

Program	PRIOR
flex 2.5.319(a)	2.51 sec
flex 2.5.319(b)	search failed (364 min)
flex 2.5.319(c)	search failed (274 min)
grep 2.5	8.83 sec
grep 2.5.1 (a)	2.59 sec
grep 2.5.1 (b)	search failed (4 min 28 sec)
grep 2.5.1 (c)	4.46 sec
make 3.80 (a)	26.92 sec
make 3.80 (b)	30 min 37 sec
bc-1.06	0.49 sec
tar-1.13.25	2.83 sec
tidy	0.90 sec
s-flex-v4	8.76 sec
s-flex-v5	search failed (96 min 20 sec)
s-flex-v6	search failed (3 min 56 sec)
s-flex-v7	3.34 sec
s-flex-v8	34.35 sec
s-flex-v9	34.51 sec
s-flex-v10	2.76 sec
s-flex-v11	2.56 sec

The results of using critical predicates to produce fault candidate sets will be introduced in later sections in this chapter.

4.3 Multiple Points Dynamic Slices: Dynamic Chops

Three different ways of computing fault candidate sets have been described in the preceding section. The faulty code is captured by all the produced fault candidate

set. Therefore it follows that if more than one kind of dynamic slice is available, the size of the fault candidate set can be further reduced by intersecting the single point dynamic slices. First Figure 4.7 shows that intersecting the forward and backward dynamic slices produces a *dynamic chop*. The dynamic chop captures the faulty

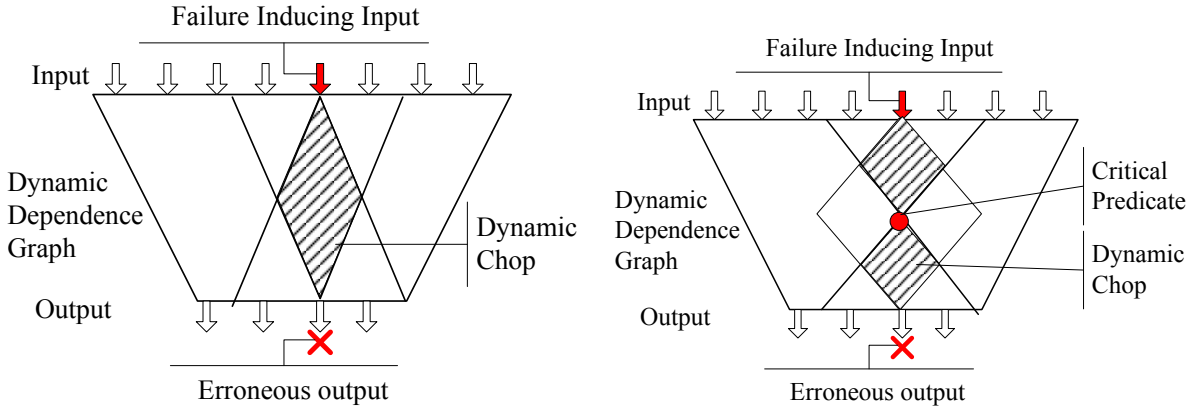


FIGURE 4.7. Multiple points dynamic slices: dynamic chop (left); and bidirectional dynamic chop (right).

code and is smaller than both the forward and backward dynamic slices. Next, if a bidirectional dynamic slice is available, it can be intersected with the dynamic chop to further reduce the size of the fault candidate set. As shown in the figure the result of this operation is a pair of dynamic chops, which is refer to as a *bidirectional dynamic chop*, one between the failure inducing input difference and a critical predicate and the other between a critical predicate and the erroneous output value.

Let us demonstrate multiple points dynamic slices using an example. The code segments in Figure 4.8 are taken from the faulty version $v4$ of program *flex* from the Siemens suite's website [2]. The assignment at statement 2983 is wrong. All the three types of slicing criteria are identified: the *minimal failure inducing input difference* is the program arguments of '`-F`' and '`-C`'; the *critical predicate* is at statement 2978; and the *erroneous output* is observed at statement 3022. The backward dynamic slice, forward dynamic slice, and bidirectional dynamic slice on the corresponding slicing criteria are computed as follows:

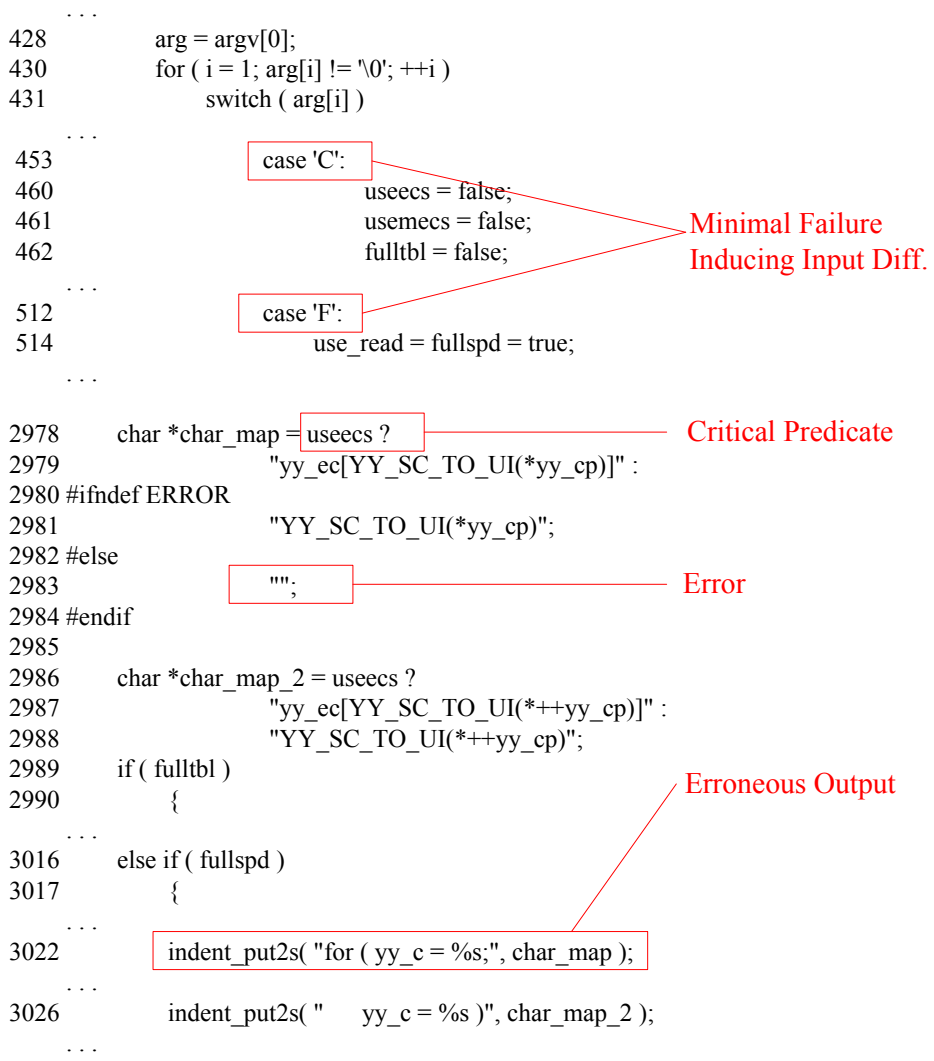


FIGURE 4.8. An example for multiple points dynamic slices – *flex v4*

$BwS = \{428, 430, 431, 453, 460, 462, 512, 514, 2978, 2979, 2983, 2989, 3016, 3022\}$
$FwS = \{431, 453, 460, 461, 462, 512, 514, 2978, 2979, 2983, 2986, 2987, 2988, 2989, 3016, 3022, 3026\}$
$BiS = \{428, 430, 431, 453, 460, 2978, 2979, 2981, 3022\}$.

They have the sizes of 14, 17, and 9, respectively. The dynamic chop and bidirectional dynamic chop are computed by taking intersections:

$BwS \cap FwS = \{431, 453, 460, 462, 512, 514, 2978, 2979, 2983, 2989, 3016, 3022\}$
$BwS \cap FwS \cap BiS = \{431, 453, 460, 2978, 2979, 2981, 3022\}$.

They contain 12 and 7 statements respectively. Statements 428 and 430 are removed from BwS by intersecting with FwS. In other words, they are not fault candidates because they are not affected by the failure inducing input. Intersecting BiS with the previous computed dynamic chop further removes 5 statements from the fault candidate set. This example clearly illustrates that multiple points dynamic slicing is very effective in reducing the size of the fault candidate set. The experimental results presented later also reveal the same fact.

4.4 Implementation

The Trimaran compiler infrastructure [4] used in earlier chapters was designed for the research of explicit instruction level parallelism. Its usability as a basis of dynamic analyses research is hence limited. For example, it does not generate native executables. Instead, it generates intermediate code that is only executable in a simulator provided by Trimaran. In order to apply dynamic slicing to a larger set of realistic programs, the implementation in this chapter is based on the *Valgrind* [1] system, which takes Intel x86 binaries generated from a *gcc* compiler.

System usage. During the execution, dynamic dependences are identified and the dynamic dependence graph is constructed. After the execution reaches its end, either because the program terminates or because the program crashes, the user is

presented with a simple debugging interface which provides limited capabilities including the ability to request computation of a dynamic slice for an execution instance of an instruction that writes to a register, writes to a memory location, or represents execution of a predicate. The slicing criteria used for a backward dynamic slice is identified by the user and input into the system. The slicing criteria for forward dynamic slicing is computed separately and then input into the system by the user. The slicing criteria for computing a critical predicate is automatically determined by the tool. Once the slicing criteria are known to the system, then through the traversals of the dynamic dependence graphs already available to the system, the computation of dynamic slices and their intersections is performed. Even though the system works on binary level, the slices are mapped back to source code level using the debugging information generated by *gcc*. For the library code, if debug information is not available, the slice is reported in terms of binary instructions. However, if the source code of the library is available, it can be recompiled with the debug option and then we can also report the portion of the slice from the library code at source code level.

Implementation details. The main components of the system carry out the following functions. The *static analysis* component of the system computes static control dependence information required during slice computations from the binary. The static analysis was implemented using the *Diablo* [3] retargetable link-time binary rewriting framework as this framework already has the capability of constructing the control flow graph from x86 binary. The *dynamic profiling* component of the system which is based upon the *Valgrind* [1] that accepts the same *gcc* generated binary, dynamically instruments it by calling the *slicing instrumenter*, and executes the instrumented code with the support of the *slicing runtime*. The slicing instrumenter and slicing runtime were developed to enable collection of dynamic information and computation of dynamic slices. Valgrind's kernel is a dynamic instrumenter which takes the binary and before executing any new (never instrumented) basic blocks

it calls the instrumentation function, which is provided by the slicing instrumenter. The instrumentation function instruments the provided basic block and returns the new basic block to the Valgrind kernel. The kernel executes the instrumented basic block instead of the original one. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The *slicing runtime* essentially consists of a set of call back functions for certain events (e.g., entering functions, accessing memory, binary operations, predicates etc.). The static control dependence information computed by the static analysis component is represented based on the virtual addresses which can be understood by Valgrind.

4.5 Experimental Evaluation

Experiments were conducted to study several issues. The first issue was about the applicability of the three slicing techniques: *Backward Slice* (BwS); *Forward Slice* (FwS); and *Bidirectional Slice* (BiS). Next the dynamic slice sizes were compared to see which technique is more effective in narrowing the size of the fault candidate set. Finally an experiment was conducted to study the synergy among these techniques, i.e. benefit of using *multiple points dynamic slicing* which narrows the fault candidate set by intersecting the resulting BwS, FwS, and BiS dynamic slices. In the remainder of this section the detailed results will be presented and analyzed about the above experiments.

4.5.1 Applicability

The applicability of the three types of dynamic slices was first studied. The results are provided in Table 4.4 where an entry of $\surd$ and $\times$ indicate whether the particular type of dynamic slice could be computed or could not be computed respectively. In addition, the presence of $\surd$ indicates that the faulty code was indeed captured by the computed dynamic slice. In other words, although in general a dynamic slice may or

may not capture the faulty code, for the failures studied the computed dynamic slices did contain the faulty code. Thus the slicing criteria used in computing the various types of dynamic slices are highly effective.

TABLE 4.4. Applicability of dynamic slice types.

Program	LOC	EXEC (%LOC)	BwS	FwS	BiS
flex 2.5.31 (a)	26,754	1871(6.99%)	√	√	√
(b)	26,754	2198 (8.22%)	√	√	×
(c)	26,754	2053 (7.67%)	√	√	×
grep 2.5	8,581	1157 (13.48%)	×	√	√
grep 2.5.1 (a)	8,587	509 (5.93%)	×	√	√
(b)	8,587	1123 (13.08%)	×	√	×
(c)	8,587	1338 (15.58%)	×	√	√
make 3.80 (a)	29,978	2277 (7.60%)	√	√	√
(b)	29,978	2740 (9.14%)	√	√	√
gzip-1.2.4	8,164	118 (1.45%)	√	√	√
ncompress-4.2.4	1,923	59 (3.07%)	√	√	√
polymorph-0.4.0	716	45 (6.29%)	√	√	√
tar-1.13.25	25,854	445 (1.72%)	√	√	√
bc-1.06	8,288	636 (7.67%)	√	√	√
tidy-34132	31,132	1519 (4.88%)	√	√	√
s-flex-v4	12,418	1631 (13.13%)	√	√	√
s-flex-v5	12,418	1882 (15.16%)	√	√	×
s-flex-v6	12,418	424 (3.41%)	√	×	×
s-flex-v7	12,418	2045 (16.47%)	√	√	√
s-flex-v8	12,418	610 (4.91%)	×	×	√
s-flex-v9	12,418	1396 (11.24%)	√	√	√
s-flex-v10	12,418	1683 (13.55%)	√	√	×
s-flex-v11	12,418	1749 (14.08%)	√	×	√

It is also observed that while each dynamic slicing technique was applicable in well over half of the 23 failures studied, there were few failures for which each of the technique could not be applied. In particular, BwS could not be computed for 5 of the 23 failures, FwS could not be computed for 3 of the 23 failures, and BiS could not be computed for 6 of the 23 failures. In addition, it is observed that while there was no failure for which all three techniques could not be applied, there were

failures where only one type of slicing technique was found to be applicable. It is also shown that in many cases all three techniques were applicable. In conclusion, the results of the experiments show that although a single dynamic slicing technique may not be applicable to all failures, it is highly likely there exist other dynamic slicing techniques taking advantage of different types of evidences. And, it is often the case that multiple slicing techniques can be used in combination.

Let us briefly discuss the reasons for each of the dynamic slicing techniques not being applicable as observed in the failed runs studied.

- (No output.) Backward slicing was found to be not applicable for *grep*. As explained in previous sections, the faults present in *grep* caused the program to terminate without the execution of any output producing statement (although the correct output is not no output). Since backward slice is computed starting at the execution instance of a statement that produced an incorrect output value, for failing runs of *grep*, no backward slicing criteria was available.

- (No failure inducing input.) Forward slice containing the root cause could not be computed for $s - flex$ versions *v6*, *v8*, and *v11*. We observed two situations in these programs. First, in some cases the fault was in form of an incorrect constant assignment (i.e., the constant value used was incorrect). As a result, the program failed on any input and the error did not appear in any computed forward slice. In other words, the failure is not induced by input. Second situation observed is as follows. To identify the failure inducing input, according to the delta debugging algorithm by Zeller [83], it is required to begin with two known inputs: one on which the program fails and another on which the program runs successfully. Unfortunately the fault was such that there was no input in the given test suite for which the program did not fail. Therefore the delta debugging algorithm could not be applied successfully.

- (No critical predicate.) Bidirectional slice could not be computed for some

versions of *flex 2.5.31*, *grep 2.5.1*, and $s - flex$. This is because a critical predicate could not be identified. The reason why sometimes a critical predicate cannot be found is because none exists. For example, if the bug is complex, simple switching of a single branch outcome may not result in the correct output being generated.

Thus, there are different situations under which each of the above technique fails. Therefore to enable the use of dynamic slicing in fault location across a broad range of programs and faults it is important to consider multiple types of dynamic slices.

4.5.2 Dynamic Slice Sizes

If dynamic slicing is not used, the programmer must search all statements executed during a failed run for faulty code. However, by employing dynamic slicing the size of the fault candidate set can be reduced from the set of executed statements. In this section the degree to which this reduction is achieved will be evaluated for the three kinds of slices (BwS, FwS, and BiS). The sizes of these three types of slices will also be compared to determine if one kind is preferable over the other types.

Lets consider the results presented in Table 4.5. In this table LOC is the lines of code in each program while EXEC is the lines of code that were executed during the failing run being used to locate faulty code. The lines of code belonging to each of the dynamic slices (BwS, FwS and BiS) are also given. Finally the MIN column gives the type and size of the smallest of the three kinds of slices.

First it can be observed that even though the executed lines of code EXEC as a percentage of total lines of code LOC ranges from only 1.45% to 16.47%, EXEC is still quite large (> 1000 in 15 out of 23 cases). Therefore reduction in the size of the fault candidate set is highly desirable. Second the data shows that the dynamic slicing techniques give significant reductions. The BwS, FwS, and BiS exceed a thousand in only 1, 4, and 2 cases respectively. In parentheses the slices sizes as a percentage

of the executed statements are also given. It can be observed that all three kinds of dynamic slices, when applicable, reduce the size of the fault candidate set quite significantly. The size of BwS ranges from 2.39% to 47.08% of EXEC. The size of FwS ranges from 0.90% to 63.18% of EXEC. Finally the size of BiS ranges from 1.54% to 60.25% of EXEC.

The *fault candidate set* metric $FCS(xS)$ indicates the fraction of lines of code that are in the fault candidate set across all the benchmarks when slice of the type xS is used. It is the ratio of the total lines of code when a slice of type xS is used ($\sum Size(xS)$) to the total lines of code that are being executed ($\sum EXEC$). However, since all kinds of slices can not be computed in all executions, the executions are differentiated into those where the slice can be computed ($App(xS)$) and those where it cannot be computed ($\overline{App(xS)}$). The full formula thus reads:

$$FCS(xS) = \frac{\sum_{App(xS)} Size(xS) + \sum_{\overline{App}(xS)} EXEC}{\sum_{All} EXEC}$$

From the data in Table 4.5 one can find that *FCS(BwS) = 0.3781*, *FCS(FwS) = 0.4919*, and *FCS(BiS) = 0.4920*. Hence, using backward slices (BwS), on average, a programmer needs to look at 37.8% of the executed statements. For forward and bidirectional slices this number is just over 49%. Thus according to this metric all three techniques are very effective.

From the MIN column in Table 4.5 it is observed that no one type of dynamic slice is consistently the smallest. Out of 18 cases for which a backward dynamic slice were computed, in 8 the backward dynamic slice is the best choice (i.e., it is the smallest). Out of 20 cases where a forward dynamic slice was computed, in 9 cases the forward dynamic slice is the smallest. Finally, out of 17 cases where a bidirectional slice was computed, in 6 it is the smallest. From this perspective the forward dynamic slice is slightly better than backward dynamic slice which is in turn slightly better than the bidirectional dynamic slice. From the MIN column it can also be observed that

in only 1 case the smallest slice size exceeds a thousand and in 8 cases the slice size is no more than 50. The numbers in parentheses are slice sizes as a percentage of LOC. As shown by the data, this number is quite small ranging from 0.04% to 6.98%. Given that the numbers in this range are quite small, it is worth pointing out that a substantial part of the reduction is the result of many statements in the program not being executed at all during the failed runs.

TABLE 4.5. Comparison of dynamic slice sizes.

Program	BwS (%EXEC)	FwS (%EXEC)	BiS (%EXEC)	MIN	(%LOC)
flex 2.5.31 (a)	695 (37.15%)	605 (32.36%)	225 (12.03%)	BiS:	225 (0.84%)
(b)	272 (12.37%)	257 (11.69%)	-	FwS:	257 (0.96%)
(c)	50 (2.44%)	1368 (66.63%)	-	BwS:	50 (0.19%)
grep 2.5	-	731 (63.18%)	88 (7.61%)	BiS:	88 (1.03%)
grep 2.5.1 (a)	-	32 (6.29%)	111 (21.81%)	FwS:	32 (0.37%)
(b)	-	599 (53.34%)	-	FwS:	599 (6.98%)
(c)	-	12 (0.90%)	453 (33.86%)	FwS:	12 (0.14%)
make 3.80 (a)	981 (43.08%)	1239 (54.41%)	1372 (60.25%)	BwS:	981 (3.27%)
(b)	1290 (47.08%)	1646 (60.07%)	1436 (52.41%)	BwS:	1290 (4.30%)
gzip-1.2.4	34 (28.81%)	3 (2.54%)	39 (33.05%)	FwS:	3 (0.04%)
ncompress-4.2.4	18 (30.51%)	2 (3.39%)	30 (50.85%)	FwS:	2 (0.10%)
polymorph-0.4.0	21 (46.67%)	3 (6.67%)	22 (48.89%)	FwS:	3 (0.42%)
tar-1.13.25	105 (23.60%)	202 (45.39%)	117 (26.29%)	BwS:	105 (0.41%)
bc-1.06	204 (32.07%)	188 (29.56%)	267 (41.98%)	FwS:	188 (2.27%)
tidy-34132	554 (36.47%)	367 (24.16%)	541 (35.62%)	FwS:	367 (1.18%)
s-flex-v4	39 (2.39%)	877 (53.77%)	37 (2.27%)	BiS:	37 (0.30%)
s-flex-v5	692 (36.77%)	1187 (63.07%)	-	BwS:	692 (5.57%)
s-flex-v6	156 (36.79%)	-	-	BwS:	156 (1.26%)
s-flex-v7	243 (11.88%)	910 (44.50%)	836 (40.88%)	BwS:	243 (1.96%)
s-flex-v8	-	-	280 (45.90%)	BiS:	280 (2.25%)
s-flex-v9	236 (16.91%)	535 (38.32%)	230 (16.48%)	BiS:	230 (1.85%)
s-flex-v10	727 (43.20%)	970 (57.66%)	727 (43.20%)	BwS:	727 (5.85%)
s-flex-v11	102 (5.83%)	-	27 (1.54%)	BiS:	27 (0.22%)

4.5.3 Multiple Points Dynamic Slices

In the preceding section a relative evaluation of the three types of dynamic slices was carried out. In contrast in this section the synergy of these techniques is studied, which is essentially the motivation for the *multiple points dynamic slicing* that was proposed earlier. In particular, the goal of this experiment is to study whether multiple points dynamic slices are significantly smaller than the individual dynamic slices.

Dynamic Chops. First the sizes of multiple points dynamic slices were obtained by intersecting the backward dynamic slice with the forward dynamic slice. The resulting data is given by the column labeled *Dynamic Chop* in Table 4.6. The size of this multiple points dynamic slice with the size of the smaller of the BwS and FwS dynamic slices which is given in column labeled $min(BwS, FwS)$ in the table. Only this data is provided for those cases where both BwS and FwS were applicable since only in these cases can a dynamic chop be computed. From the data in Table 4.6 one can tell that the size of this dynamic chop can be significantly smaller than the size of the smaller of the BwS and FwS slices. The numbers in parentheses give the sizes of the dynamic chops as a percentage of the sizes of $min(BwS, FwS)$. As shown in the table, in 7 cases out of 13, the size of the dynamic chop slice is less than half the size of the smaller of the BwS and FwS dynamic slices.

TABLE 4.6. Sizes of dynamic chops and bidirectional dynamic chops.

Program	min(BwS,FwS)	Dynamic Chop		min(BwS,FwS,BiS)	Bidirectional Dynamic Chop	
flex 2.5.31 (a)	605	256	(42.31%)	225	27	(12.00%)
(b)	257	102	(39.69%)	-	-	
(c)	50	5	(10.00%)	-	-	
grep 2.5	-	-		88	86	(97.73%)
grep 2.5.1 (a)	-	-		32	25	(78.13%)
(b)	-	-		-	-	
(c)	-	-		12	12	
make 3.80 (a)	981	739	(75.33%)	981	739	(75.33%)
(b)	1290	1104	(85.58%)	1290	1051	(81.47%)
gzip-1.2.4	3	3		3	3	
ncompress-4.2.4	2	2		2	2	
polymorph-0.4.0	3	3		3	3	
tar-1.13.25	105	103	(98.09%)	105	45	(42.86%)
bc-1.06	188	102	(54.26%)	188	102	(54.26%)
tidy-34132	367	164	(44.69%)	367	161	(43.87%)
s-flex-v4	39	7	(17.95%)	37	7	(18.92%)
s-flex-v5	692	544	(78.61%)	-	-	
s-flex-v6	-	-		-	-	
s-flex-v7	243	63	(25.93%)	243	63	(25.93%)
s-flex-v8	-	-		-	-	
s-flex-v9	236	112	(47.46%)	230	112	(47.46%)
s-flex-v10	727	574	(78.95%)	727	574	(78.95%)
s-flex-v11	-	-		27	27	

Bidirectional Dynamic Chop. Second the sizes of multiple points dynamic slice were obtained by intersecting all three dynamic slices: BwS, FwS, and BiS. The resulting data is given by the column labeled *Bidirectional Chop* in Table 4.6. The sizes of bidirectional dynamic chops are compared with the sizes of the smallest slices of the BwS, FwS, and BiS dynamic slices which are given in the column labeled $min(BwS, FwS, BiS)$. This data is only provided for those cases where all three (BwS, FwS, and BiS) were applicable. From the data in Table 4.6 one can see that the sizes of the bidirectional dynamic chops can be significantly smaller than the sizes of the smallest slices of the BwS, FwS, and BiS dynamic slices. The numbers in parentheses give the sizes of bidirectional dynamic chops as a percentage of the sizes of $min(BwS, FwS, BiS)$. As one can see in 6 cases out of 11, the size of the bidirectional dynamic chop is less than half the size of the smallest of the BwS, FwS, and BiS dynamic slices. Therefore bidirectional dynamic chopping is a very promising technique.

4.5.4 Discussion

Finally lets summarize the benefits that the dynamic slicing techniques studied provide to the programmer in carrying out fault location. Table 4.7 summarizes the number of lines in the fault candidate set (FCS) produced using the techniques described earlier which when compared to the total lines of code (LOC) in the test programs is very small. The numbers in parentheses are the sizes of FCS as a percentage of LOC. As one can see, in 15 cases this percentage is no more than 1%. In all other cases it is a few percent. However, in a few cases the FCS contains a significant number of statements. In these cases the statements contained can be ranked according to their dependence distances from the erroneous output as proposed in [52, 88]. As a study in [88] illustrated, such ranking leads to the user having to examine less than half of the statements in a backward dynamic slice. Finally, fault location tech-

TABLE 4.7. Summary of dynamic slice sizes.

Program	LOC	FCS	(%LOC)
flex 2.5.31 (a)	26,754	27	(0.10%)
(b)	26,754	102	(0.38%)
(c)	26,754	5	(0.02%)
grep 2.5	8,581	86	(1.00%)
grep 2.5.1 (a)	8,587	25	(0.29%)
(b)	8,587	599	(6.98%)
(c)	8,587	12	(0.14%)
make 3.80 (a)	29,978	739	(2.47%)
(b)	29,978	1051	(3.51%)
gzip-1.2.4	8,164	3	(0.04%)
ncompress-4.2.4	1,923	2	(0.10%)
polymorph-0.4.0	716	3	(0.42%)
tar-1.13.25	25,854	45	(0.17%)
bc-1.06	8,288	102	(1.23%)
tidy-34132	31,132	161	(0.52%)
s-flex-v4	12,418	7	(0.06%)
s-flex-v5	12,418	544	(4.38%)
s-flex-v6	12,418	156	(1.26%)
s-flex-v7	12,418	63	(0.51%)
s-flex-v8	12,418	280	(2.25%)
s-flex-v9	12,418	112	(0.90%)
s-flex-v10	12,418	574	(4.62%)
s-flex-v11	12,418	27	(0.22%)

TABLE 4.8. Data slices (DS) and backward dynamic slices (BwS).

Program	DS (Exec%)	BwS (Exec%)
gzip-1.2.4	14 (11.86%)	34 (28.81%)
ncompress-4.2.4	13 (22.03%)	18 (30.51%)
polymorph-0.4.0	17 (37.78%)	21 (46.67%)
tar-1.13.25	44 (9.89%)	105 (23.60%)
bc-1.06	76 (11.95%)	204 (32.07%)
tidy-34132	148 (9.74%)	554 (36.47%)

niques such as those presented eventually require a programmer to spend efforts in understanding the cause of a failure and correcting the faulty code. This effort can be reduced by using dynamic slicing based approach because not only is the programmer able to examine faulty code but also the statements on which the faulty code depends and the statements that depend upon the faulty code. Examining the dependence relationships is very helpful in understanding the cause of the failure. Finally it is worth mentioning that the examples shown in earlier sections, which were taken from the failures studied in the experiments, illustrated that locating the fault by examining the dynamic slices can be quite easy in some cases. Additional examples from the considered bugs also illustrating a similar behavior can be found in case studies presented in [84].

4.6 Other Types of Dynamic Slices

There are other types of dynamic slices than the ones discussed thus far. All the previous discussed dynamic slices consider both data dependences and control dependences during computation. If only data dependences are considered and the DDG is traversed backward, the resulting slices are called *dynamic data slices* (DS) [87, 88].

It is easy to identify that a program has been affected by a memory bug because it crashes with a segmentation fault error. The programmer can use dynamic data slicing instead of traditional backward dynamic slicing in such situations. The reason

why data slices are so effective for memory bugs is that the program crash is caused by the presence of an *unexpected dynamic data dependence* between the point at which memory is corrupted and the later point at which the corrupted value is used. In fact the memory corruption typically corrupts a pointer and its use causes a crash because it dereferences the pointer. Dynamic data slice captures all appropriate dynamic data dependences including the unexpected dynamic data dependence and therefore it is able to capture faulty code. To illustrate the above, let us review the example of `gzip` in Figure 4.2. Since the unexpected dynamic data dependence from statement 40, where variable `env` was defined by mistake, to statement 1344, where variable `env` was used, is captured by the data slice, the bug can be easily located without considering any control dependences. Data slices have been computed for the memory bugs in Table 2.3 presented in chapter 2. All the memory bugs were captured by data slices. In addition, as shown in Table 4.8, data slices are significantly smaller than *backward dynamic slices*. Thus, data slices instead of full backward dynamic slices ought to be computed when a bug can be identified as a memory one.

A backward dynamic slice may not be able to capture the error even though the wrong output is actually caused by the error. Figure 4.9 gives an example. It is taken from the *gzip* version three provided by the website [2] of Siemens suite [43]. The error is in the assignment to *save_orig_name*. The correct code is *save_orig_name=!no_name*. In the failed run, since save_orig_name contained the wrong value $False$, branch S3 was not taken such that $flags$ had the wrong value 0 while it should have been defined as $ORIG\_NAME$ at S3. This wrong $flags$ value was finally propagated to the output file. The backward dynamic slice BwS of the wrong output does not contain the error because S4 depends on S2. The fact that S4 could have had a different value if S3 had taken the other branch can not be captured by the backward dynamic slicing technique itself.

In such a case, a relevant slice is [33] computed instead which is larger than the backward dynamic slice. In *relevant slicing*, a *potential dependence* is introduced

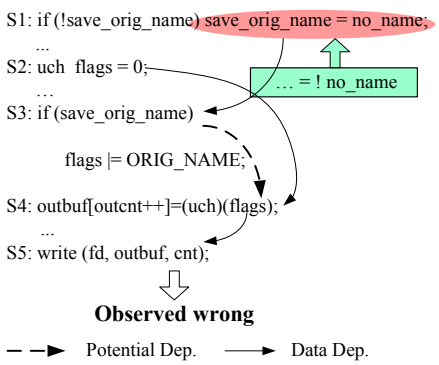


FIGURE 4.9. Gzip v3 r1

between S4 and S3 such that S1 is reachable from the wrong output. However, these potential dependence edges are introduced for each node in DDG which can result in a much larger slice. Fortunately, in the real bugs studied relevant slices were not needed and the backward dynamic slices were effective enough to capture all of them.

In general, the same fact that certain code did not get executed while it should have due to the bug may result in other types of the previously discussed dynamic slices such as forward slices and bidirectional slices precluding the root cause as well [82].

4.7 Summary

In this chapter, two new types of dynamic slices were introduced – forward slices on minimal failure inducing input differences and bidirectional slices on critical predicates. These new types of dynamic slices, together with the traditional backward dynamic slices, broaden the applicability of dynamic slicing. Because while all types of dynamic slices may not be applicable in certain situations, it is highly unlikely none of them is applicable. In addition, if multiple types of dynamic slices can be computed for a failed run, a much smaller fault candidate set can be produced by intersecting them. In 15 out of the total 23 bugs under study, the fault candidate set is less than 1% of the lines of code. In 6 out of the 11 bugs for which the BwS,

FwS, and BiS are all available, the intersection of the three types of slices, or the bidirectional dynamic chop, is less than 50% of the smallest of the three types of slices. In the subsequent chapters, it will be shown how value profiles can also be collected and used to further reduce the fault candidate sets.

Chapter 5
Efficiency of Value Profiles

Earlier chapters discussed an efficient representation of dependence profiles as well as its use in generating fault candidate sets. Next the exploitation of value profiles will be considered. In this chapter, an efficient representation for value profiles is developed and in the next chapter it will be shown how value profiles can be used to produce smaller fault candidate sets. Value profiles are compressed using a two tier strategy. First, redundancy is removed from the value profiles in a similar fashion to dependence profiles. Second, a generic stream compression technique is developed which provides both a high compression rate and the feature of bidirectional traversibility.

5.1 Removing Redundancy in Value Profiles

It is well known that subcomputations within a program are often performed multiple times on the same operand values – this observation is the basis for widely studied techniques for reuse based redundancy removal [70]. Next the same observation can be exploited in devising a compression scheme for sequence of values associated with statements belonging to a node in the WET.

The compression scheme can be illustrated using the example below in which the value of x is an input to a node and using this value, the values of y and z are computed. Further assume that while the node is executed four times, only two unique values of x (x_1 and x_2) are encountered in the value sequence $Vals[0..3] = [x_1 x_2 x_1 x_2]$. Given the nature of the computation, the values of y and z also follow similar patterns. The value sequences can be compressed by storing each unique value produced by a statement only once in the $UVals[0..1]$ array. In addition, the proposed scheme remembers the pattern in which these unique values are encountered. This pattern

is of course common to the entire group of statements. The pattern [0101] gives the indices of values in the $UVals[]$ array that are encountered in each position. Clearly the $Vals[0..3]$ corresponding to each statement can be determined using the following relationship.

$$Values[i] = UValues[Pattern[i]]$$

Before

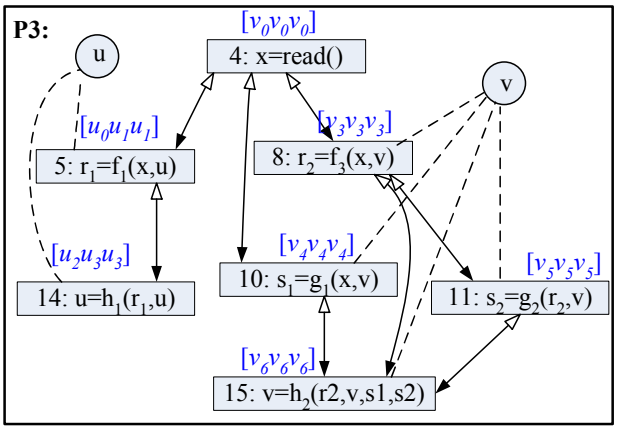
Statement	$Vals[0..3]$
x	$[x_1 x_2 x_1 x_2]$
$y = f(x)$	$[y_1 y_2 y_1 y_2]$
$z = g(x, y)$	$[z_1 z_2 z_1 z_2]$

After: Pattern=[0101]

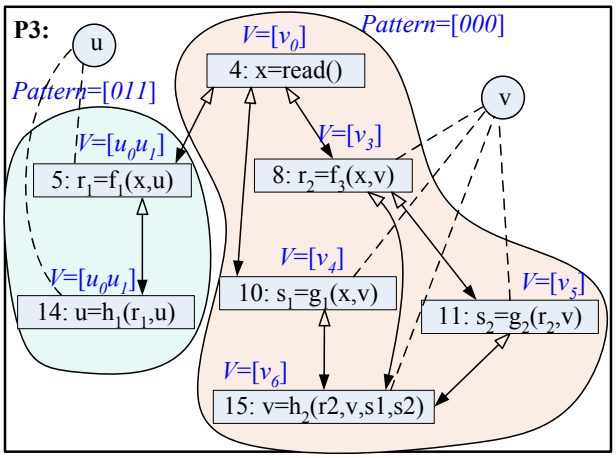
Statement	$UVals[0..1]$
x	$[x_1 x_2]$
$y = f(x)$	$[y_1 y_2]$
$z = g(x, y)$	$[z_1 z_2]$

The above technique yields compression because by storing the pattern only once, it becomes possible to eliminate all repetitions of values in value sequences associated with all statements. The ease with which the sequence of values can be generated from the unique values is a good characteristic of this compression scheme. The compression achieves space savings at the cost of slight increase in the cost of recovering the values from WET.

In the above discussion the situation considered is such that all of the statements shared a single pattern. In general, multiple patterns may be desirable because different subsets of statements may depend upon different subsets of inputs that are either received from outside the node or are read through input statements within the node. Statements belonging to a node are subdivided into disjoint groups as follows. For each statement the input variables that it depends upon (directly or indirectly) is first determined. Groups are first formed by including all statements that depend upon exactly the same inputs into the same group. Next if a group depends upon set of inputs that are a proper subset of inputs for another group, then the two groups are merged. Finally input statements within the node on which many groups depend is included in exactly one of the groups. Once the groups are formed, for each group a pattern is found and the values are compressed according to the groups pattern.



(a) Before compression.



(b) After compression.

FIGURE 5.1. Value compression.

In Figure 5.1 formation of groups for node $P3$ is illustrated. The first figure shows the value sequences associated with statements before compression. The statements depend upon values of u and v from outside the node and the value of x that is read by a statement inside the node. Two groups are formed because some statements depend upon values of x and v while other statements depend upon values of x and u. The statement that reads the value of x is added to one of the groups. Once the groups have been identified, patterns are formed for each group as shown.

5.2 Prediction Based Compression of Value Profiles

In the next step of compression information labeling a WET can be viewed as consisting of two streams of values arising from the sequence of $< t, v >$ pairs labeling a node: one corresponding to the timestamps (t's) and the other corresponding to the values (v's).

The stream compression algorithm should be designed such that the compressed stream of values can be rapidly traversed. An analysis algorithm using the WET representation may traverse the program representation in forward or backward direction (recall that is why all edges in WET are bidirectional). Thus, during a traversal, it is expected that the profile information, and hence the values in above streams, will be inspected one after another either in forward or backward direction. Unfortunately most of the existing algorithms for effectively compressing streams are *unidirectional*, i.e., the compressed stream can be uncompressed only in one direction typically starting from the first value and going towards the last. Examples of such algorithms include compression algorithms designed from value predictors which were used for compressing value and address traces in [16]. The problem with using a *unidirectional* predictor is that while it is easy to traverse the value stream in the direction corresponding to the order in which values were compressed, traversing the stream in the reverse direction is expensive. The only way to efficiently traverse the streams freely in both directions is to uncompress them first which is clearly undesirable. Sequitur [62] which was used for compressing control flow traces in [53] and address traces in [21] yields a representation which can be traversed in both directions. However, it is well known that Sequitur is not nearly as effective as the above unidirectional predictors when compressing value streams [16].

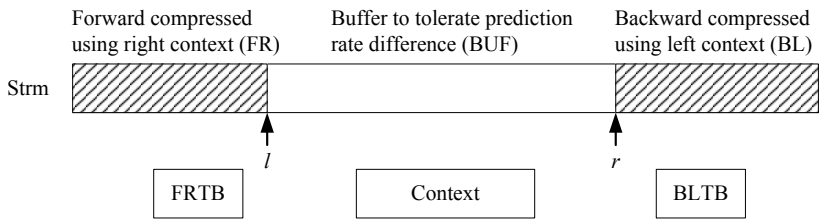
To overcome the above problem with existing compression algorithms, a novel approach to constructing *bidirectional* compression algorithms is introduced. The approach can be used to convert an *unidirectional* value predictor based compression

algorithm [16] into a *bidirectional* one. Lets consider the highly effective FCM predictor [72, 71]. A unidirectional FCM predictor compresses a stream in the *forward direction* such that a value is successfully compressed if it can be correctly predicted from its *left context* (i.e., pattern of immediately preceding n values); otherwise the value is saved in uncompressed form. A look up table TB is maintained to store predictions corresponding to a limited number of left context patterns encountered in the past. The index of the entry at which the prediction for a pattern is stored is derived by hashing the pattern into a number.

If a value is correctly predicted by the look up table TB using the left context, a bit 1 is created in the compressed stream. If a prediction v provided by the look up table TB using the left context does not match the value v' being compressed, then a bit sequence of $< \overline{v} \cdot 0 >$ is created in the compressed stream while the look up table TB is updated using v' to enable future predictions. Here $\overline{v}$ denotes the bits for v. Clearly for a stream compressed in the above fashion only forward traversal is possible.

5.2.1 Bidirectional compression derived from the FCM predictor

Now lets look at the design of a bidirectional predictor. In particular, lets look at a bidirectional counterpart of the FCM predictor [72, 71]. A bidirectional differential FCM predictor [30] can be constructed in a similar way. Normal FCM is *forward* compressed and then *forward* traversed. If the direction of table lookup is changed from using *left* context to using *right* context, which means future values are used to predict the current value instead of previous values being used to predict next value, a *forward* compressed and *backward* traversed FCM can be constructed. Similarly, a *backward* compressed and *forward* traversed FCM can be developed. A bidirectional FCM(BFCM) can be achieved by using these two FCMs back to back.



Before the introduction of the algorithms for bidirectional traversal of the value stream, lets introduce the notation. Let m be the *length* of the uncompressed value stream, n be the *context size*, a BFCM can be viewed as a tuple of $< Strm$, $FRTB$, $BLTB, i, l, r, Context >$ where:

- $Strm$ is the compressed bit stream composed of two substreams: FR and BL. FR is obtained by compressing values at positions 1 through $(i-1)$ in *forward direction* (F) using *right context* (R). BL is obtained by compressing values at positions $(i+n)$ through $(m-1)$ in the *backward direction* (B) using the *left context* (L).

- $Context$ is a buffer which contains the current context of n uncompressed values from position i to position $(i+n-1)$.

- $FRTB$ is the lookup table for FR while $BLTB$ is the lookup table for BL.

- Finally, l is the end bit position in $Strm$ of FR while r is the starting bit position of BL in $Strm$. The reason for providing extra bits (BUF) between positions l and r will be discussed in greater detail later – essentially these bits provide extra space needed to accommodate the differences between forward and backward compression rates.

There are four types of basic operations for a BFCM on which the traversal operations are built. *FORWARD_COMPRESS* compresses a value v into $Strm$ starting at

bit position l using *FRTB*. Parameter *Context* is the right context for v. The difference between this operation and the forward compressing operation in a conventional FCM is that *FORWARD_COMPRESS* uses the *right* context instead of *left* context. Using *right* context to compress *forward* provides the capability to uncompress in the *backward* direction. *BACKWARD_UNCOMPRESS* consumes bits in the backward direction starting at l, which were generated earlier by *FORWARD_COMPRESS* operation, to uncompress the value to the left of the current context. The other two operations, *FORWARD_UNCOMPRESS* and *BACKWARD_COMPRESS* can be constructed in a similar way. The details of all four operations are given in Figure 5.2.

To traverse one step forward, BFCM first forward uncompresses the value to the right of *Context*, U_{i+n}, by looking at the bits starting at $Strm_r$ and then shifts *Context* one step forward and uses the new *Context* to forward compress the value to the left. Backward traversal can also be similarly defined. The implementation of the traversal operations in terms of the four basic operations is given in Figure 5.3. Note that it is assumed that a 32 bits machine is used. Hence if a value is predicted, it consumes one bit space, if not, it consumes 32+1 bits of space.

The example in Figure 5.4 illustrates the above algorithm. The first figure shows a portion of the uncompressed stream while the second figure shows the state of the stream and look up tables corresponding to four consecutive positions of the context which consists of three uncompressed values. No matter whether the stream is traversed forwards or backwards, the sequence of states encountered is the same.

5.2.2 Accounting for the difference in forward and backward compression rates

One implementation problem arises due to different prediction rates of the two FCMs. As a result the amount of space needed to store the stream will vary at different points of the traversal. To handle this problem the design goal is to allocate enough extra

Basic Operations

FORWARD_COMPRESS$(v, Strm, l, FRTB, Context)$
(1) $index = hash(Context)$
(2) `if` $FRTB[index] = v$ `then`
(3) $\quad Strm_{l...l+1} =< 1 >$
(4) $\quad l = l + 1$
(5) `else`
(6) $\quad Strm_{l...l+33} =< \overline{v} \cdot 0 >$
(7) $\quad l = l + 33$
(8) $\quad FRTB[index] = v$
(9) `endif`

FORWARD_UNCOMPRESS$(Strm, r, BLTB, Context)$
(1) $b = Strm_r$
(2) $r = r + 1$
(3) $index = hash(Context)$
(4) `if` $b = 1$ `then`
(5) $\quad v = BLTB[index]$
(6) `else`
(7) $\quad v = Strm_{r...r+32}$
(8) $\quad r = r + 32$
(9) $\quad BLTB[index] = v$
(10) `endif`
(11) `return` v

BACKWARD_COMPRESS$(v, Strm, r, BLTB, Context)$
(1) $index = hash(Context)$
(2) `if` $BLTB[index] = v$ `then`
(3) $\quad Strm_{r-1...r} =< 1 >$
(4) $\quad r = r - 1$
(5) `else`
(6) $\quad Strm_{r-33...r} =< 0 \cdot \overline{v} >$
(7) $\quad r = r - 33$
(8) $\quad BLTB[index] = v$
(9) `endif`

BACKWARD_UNCOMPRESS$(Strm, l, FRTB, Context)$
(1) $b = Strm_l$
(2) $l = l - 1$
(3) $index = hash(Context)$
(4) `if` $b = 1$ `then`
(5) $\quad v = FRTB[index]$
(6) `else`
(7) $\quad v = Strm_{l-32...l}$
(8) $\quad l = l - 32$
(9) $\quad FRTB[index] = v$
(10) `endif`
(11) `return` v

FIGURE 5.2. Four basic operations used by BFCM.

Traverse
– $< Strm, FRTB, BLTB, Context, i, l, r >$ is the bit stream to traverse;
$STEP\_FORWARD(Strm, FRTB, BLTB, Context, i, l, r)$
(1) $v = FORWARD\_UNCOMPRESS(Strm, r, BLTB, Context)$
(2) $t = Context[0]$
(3) $Context = Context[1...n-1] \cdot v$
(4) $FORWARD\_COMPRESS(t, Strm, l, FRTB, Context)$
(5) $i = i + 1$
(6) `return` v

$STEP\_BACKWARD(Strm, FRTB, BLTB, Context, i, l, r)$
(1) $v = BACKWARD\_UNCOMPRESS(Strm, l, FRTB, Context)$
(2) $t = Context[n-1]$
(3) $Context = v \cdot Context[0...n-2]$
(4) $BACKWARD\_COMPRESS(t, Strm, r, BLTB, Context)$
(5) $i = i - 1$
(6) `return` v

FIGURE 5.3. Forward and backward traversal by a single step.

space so that at any point during traversal there is enough space available to handle the stream. The space allocation is performed in a manner that at any point in time the context (uncompressed values) are held in the $Context$ buffer while all other values (forward and backward compressed values) are kept in $Strm$ storage. The space allocated between l and r in $Strm$ (referred to as BUF) is there to accommodate the difference between forward and backward compression rates. For example, when the cursor moves forward or backward by one step, it is possible that the value that is uncompressed frees up one bit (i.e., the value had been compressed to one bit) while the value that is compressed requires 33 bits (i.e., the value cannot be successfully compressed). The additional bits allocated accommodate these extra bits. In fact the BUF size is computed in such a way that whenever extra space is needed it is available in BUF.

To ensure that there is sufficient extra space allocated in BUF so that forward and backward traversals never cause the compressed stream size to overflow the allocated space, an algorithm is used as described in Figure 5.5. This algorithm first *forward* compresses all the values into a temporary bit stream Tmp. However, Tmp

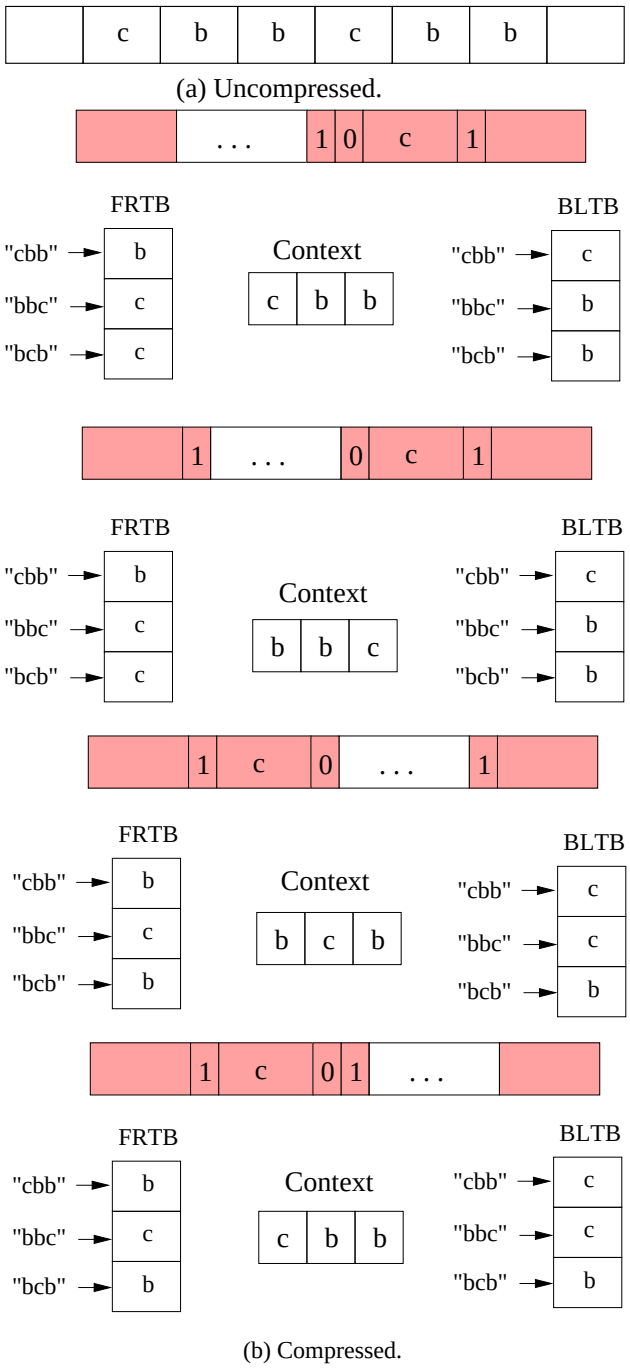


FIGURE 5.4. Example of bidirectional FCM compression.

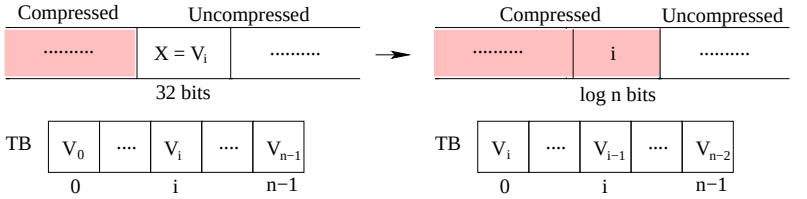
is not bidirectional traversable yet. A backward traversal is performed to determine the amount of additional space that needs to be allocated. Lines 13 to 17 in the algorithm pre-allocate extra space. Condition $r < l + 33$ being true means that the next *BACKWARD_COMPRESS* operation may overwrite the bits generated by *FORWARD_COMPRESS* operation previously, in other words, both the FCMs encounter low prediction rate and then the allocated space may not be enough. In this case, BFCM inserts some buffer space between FR and BL. After *backward* traversing Tmp once with allocating buffer space to tolerate different prediction rates, the BFCM $< Strm, FRTB, BLTB, Context, 0, l, r >$ is ready to be used for bidirectional traversal.

Compress value stream
– $Vals$ is the uncompressed stream;
– Tmp is a bit stream with $INFINITE$ length;
$COMPRESS(Vals, vLen)$
(1) $Context = Vals[0...n-1]$
(2) $sLen = 0$
(3) `for` $i = 0$ `to` $vLen - n - 1$
(4) $Context = Context[1...n-1] \cdot Vals[n+i]$
(5) $FORWARD\_COMPRESS(Vals[i], Tmp, sLen, FRTB, Context)$
(6) `endfor`
(7) $r = sLen$
(8) $l = sLen$
(9) `for` $i = vLen - n - 1$ to 0
(10) $v = BACKWARD\_UNCOMPRESS(Tmp, l, FRTB, Context)$
(11) $v_{com} = Context[n-1]$
(12) $Context = v \cdot Context[0...n-2]$
(13) `if` $r < l + 33$ `then`
(14) $Tmp_{(r+EXTRA)...(sLen+EXTRA)} = Tmp_{r...sLen}$
(15) $r = r + EXTRA$
(16) $sLen = sLen + EXTRA$
(17) `endif`
(18) $BACKWARD\_COMPRESS(v_{com}, Tmp, r, BLTB, Context)$
(19) `endfor`
(20) $Strm = Tmp_{0...sLen}$
(21) `return` $< Strm, FRTB, BLTB, Context, 0, l, r >$

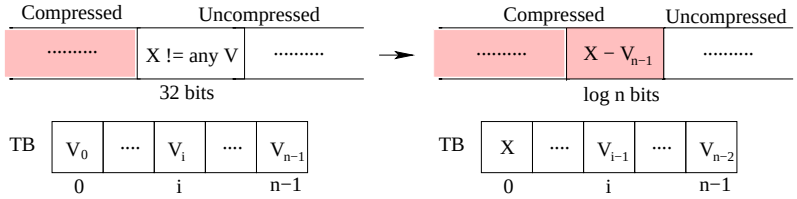
FIGURE 5.5. Preparing streams for bidirectional traversal.

5.2.3 Bidirectional compression derived from a Last n predictor

Another predictor which has been used for unidirectional compression is the last n predictor [56, 17]. A bidirectional compression algorithm is also derived using the last n predictor. This is because studies have shown that while overall performance of both FCM and Last n predictors is quite good, there are also specific situations where one predictor works well while the other does not and vice versa [16]. The full details of bidirectional compression algorithm based upon last n predictor are omitted due to space limitations. However, the main cases of forward compression of a value are summarized in Figure 5.6. Backward compression is similar. Unlike the bidirectional FCM predictor only a single look up table TB is used for both forward and backward compression.



(a) Successful compression of ith value (forward direction).



(b) Failed compression of ith value (forward direction).

FIGURE 5.6. Bidirectional last n compression.

5.2.4 Selection

For each stream one is selected from several bidirectional versions of compression methods. Initially all methods are used to compress the stream. After a certain number of instances the method that performs the best up to that point is picked.

The implemented methods include the FCM, differential FCM (this is an adaptation of FCM that works on strides [30]), last n, and last n stride. For each type three versions are created with differing context size.

5.3 Experimental Results

The experimental setting is the same as mentioned in chapter 2. The two tier compression strategy is first evaluated for value profiles. Then the performance of the compression scheme on dependence profiles is also studied. Finally, the overall compression effect on complete WETs is evaluated.

5.3.1 Compression of Value Profiles

The first experiment is about compressing value profiles using the two tier strategy. Table 5.1 shows the sizes of node labels, timestamp and value sequences, before and after compression.

TABLE 5.1. Effect of compression on value profiles.

Benchmark	t_s labels			val_s labels		
	Orig. (MB)	Orig./ Tier-1	Orig./ Tier-2	Orig. (MB)	Orig./ Tier-1	Orig./ Tier-2
099.go	2614.12	37.96	47.13	1847.09	2.48	6.33
126.gcc	1391.60	50.06	126.63	945.03	3.15	17.62
130.li	2822.26	32.47	105.88	1894.48	3.83	17.33
164.gzip	2481.32	30.33	152.76	1733.13	1.66	4.02
181.mcf	2728.12	22.12	127.09	1875.21	2.37	7.02
197.parser	2347.92	30.61	101.82	1615.57	2.05	12.45
255.vortex	2324.87	53.51	176.55	1641.31	3.51	23.82
256.bzip2	2865.81	55.24	1171.6	2154.85	2.46	10.61
300.twolf	2633.64	27.36	69.49	1873.52	2.13	4.36
Avg.	2467.74	37.74	230.99	1731.13	2.63	11.51

The above results show that while the average compression ratio of the t_s labels is very high, which is 231, the same is not true for *value sequences* that label the nodes (compression ratio for these is only 11.5). Compression of values is much harder – even though the value compression algorithm is aggressive, the compression ratios

for value sequences are modest in comparison to those for timestamp sequences. The main reason is that not as much patterns occur in value sequences as in timestamp sequences.

5.3.2 Using Prediction Based Compression for Dependence Profiles

As described in chapter 2, dynamic dependences are represented by annotating a static dependence edge with a sequence of timestamp pairs. In chapter 3, a series of optimizations have been developed to eliminate redundancy in dependence profiles, which can ba considered as the first tier compression. After applying the optimizations, the remained sequence of $< t_{s_2}, t_{s_1} >$ on a dependence edge gives rise to two streams, one corresponding to the first timestamps (t_{s_2}'s) and the other corresponding to the second timestamps (t_{s_1}'s). Each of the above streams can be compressed using the prediction based compression technique. Table 5.2 presents the results. Here `Tier-1` denotes the optimizations introduced in chapter 3.

TABLE 5.2. Effect of compression on dependence profiles.

Benchmark	Edge labels		
	Orig. (MB)	Orig./ Tier-1	Orig./ Tier-2
099.go	5908.12	9.00	26.00
126.gcc	2901.26	15.37	118.94
130.li	5682.32	11.36	84.74
164.gzip	5473.42	10.13	60.37
181.mcf	5938.54	7.62	46.56
197.parser	4766.38	15.57	133.92
255.vortex	4781.46	21.75	212.35
256.bzip2	6900.52	32.06	455.44
300.twolf	6159.03	7.05	34.43
Avg.	5390.12	14.43	130.31

The results show that the prediction based compression provides very high compression rates. Together with the redundancy removing optimizations (Tier-1), the compression ratio is 130 on an average. The reduction of storage space is not the sole goal. The other objective is to provide easy access to the compressed profiles.

Therefore, another experiment was carried out to compare the execution times of backward dynamic slicing on the dynamic dependence graphs in their original forms (`Orig.`), after optimizations (`Tier-1`), and after optimizations and the prediction based compression (`Tier-2`). In this experiment, the prior runs were cut at the boundaries of from 114 and 139 Million intermediate level statements, which are very close to the trace lengths used in previous sections. The average times needed to compute a backward dynamic slice after tier-1 and tier-2 compression are a little over 14.34 seconds and 90.98 seconds respectively. While the optimizations in tier-1 only speed up the slice computation, the slowdown incurred by the tier-2 compression is reasonable given the gains in space savings. Note that the response times for the `099.go` benchmark are higher than other programs. Due to the complex control flow structure of `099.go` each node has several incoming edges and thus it takes longer to identify the appropriate relevant edge during traversal.

TABLE 5.3. Dynamic slicing on compressed DDGs (avg. over 25 slices).

Benchmark	Stmts Executed (Millions)	Tier-1 (sec.)	Tier-2 (sec.)	Tier-2/ Tier-1
099.go	132.52	58.31	412.44	7.07
126.gcc	139.46	10.91	17.74	1.63
130.li	126.78	10.00	121.42	12.14
164.gzip	123.06	4.20	102.33	24.34
181.mcf	137.31	17.47	76.07	4.35
197.parser	122.12	1.55	4.69	3.02
255.vortex	119.50	4.75	18.09	3.81
256.bzip2	128.25	2.76	3.90	1.42
300.twolf	114.85	19.10	62.15	3.25
Avg.	127.09	14.34	90.98	6.78

5.3.3 Overall Compression of WETs

In earlier experiments, the compression of individual type of profiles was evaluated. It would be interesting to study the overall performance of the two tier compression

strategy on complete WETs, which contain control flow, value, address, and dependence profile information. The overall effect of the two tier compression strategy is summarized in Table 5.4. While the average size of the original uncompressed WETs (Orig. WET) is 9589 Megabytes, after compression their size (Comp. WET) is reduced to 331 Megabytes which represents a compression ratio (Orig./Comp.) of 41. Therefore on an average the proposed approach enables saving of the whole execution trace corresponding to program run of 647 Million intermediate statements using 331 Megabytes of storage.

TABLE 5.4. WET sizes.

Benchmark	Input	Stmts Executed (Millions)	Orig. WET (MB)	Comp. WET (MB)	Orig./ Comp.
099.go	training	685.28	10369.32	574.65	18.04
126.gcc	ref/insn-emit.i	364.80	5237.89	89.03	58.84
130.li	ref	739.84	10399.06	203.01	51.22
164.gzip	training	650.46	9687.88	537.72	18.02
181.mcf	testing	715.16	10541.86	416.21	25.33
197.parser	training	615.49	8729.88	188.39	46.34
255.vortex	training/lendian	609.45	8747.64	104.59	83.63
256.bzip2	training	751.26	11921.19	220.70	54.02
300.twolf	training	690.39	10666.19	646.93	16.49
Avg.	n/a	646.90	9588.99	331.25	41.33

In Figure 5.7 the relative sizes of the three main components of profile data (node timestamp sequences, node value timestamp sequences, and edge timestamp sequences) are shown before compression (Original), after first tier compression (After Tier-1), and after second tier compression (After Tier-2). As shown in the table, the contribution of value sequences to the total size increases in percentage following each compression step since the degree of compression achieved for value sequences is lower.

Next the *scalability* of WETs is studied so that it can be estimated the limit on the length of a program run for which the whole execution trace can be realistically kept in memory. For this purpose the impact of trace length on the compression ratios is studied. In Figure 5.8, the executions are divided into ten intervals for each benchmark (x-axis) and then the compression ratios (y-axis) are measured up to each

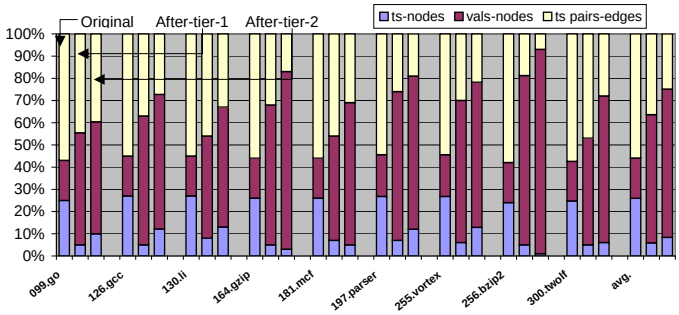


FIGURE 5.7. Relative sizes of WET components.

interval. From the results in Figure 5.8 it can be noticed that for 7 out of 9 programs the compression ratios either improve or roughly remain the same as the length of the run increases. For benchmark *256.bzip2*, a sharp decrease of the compress ratio is observed from the second interval to the third interval. It is very likely due to the switch of program phases. The new phase is substantially more difficult to compress compared to the previous one. As this phase finishes, its effect gradually fades out and the compression ratio gradually recovers.

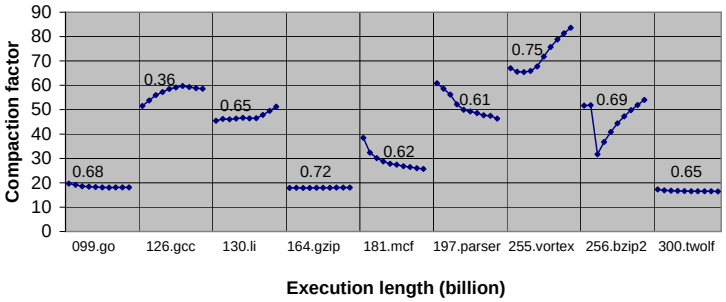


FIGURE 5.8. Scalability of compression ratio.

Lets assume that the compression ratio remains constant across the length of a program run. Further recall that earlier experiments show that the compressed WET for execution of 647 Million Trimaran intermediate code statements takes approximately 331 Megabytes of storage. Therefore it can be shown by extrapolation that the WET corresponding to a program run involving execution of 3.9 Billion Trimaran intermediate code statements consumes 2 Gigabyte of space, which is the normal RAM size for a workstation. It is a fairly long trace and thus can be used effectively in fault location and studying program behaviors when designing compilers and architectures.

The times taken to construct the compressed WETs for the program runs are

presented in Figure 5.9. Similar to the prior experiment, the executions were divided into ten intervals with equal length and then the cumulative construction times were collected up to each interval. The results show that it takes 200-300 minutes to construct the WETs fully for most of the runs depending on the execution lengths. It can also be observed that the construction time increases almost linearly with the execution length.

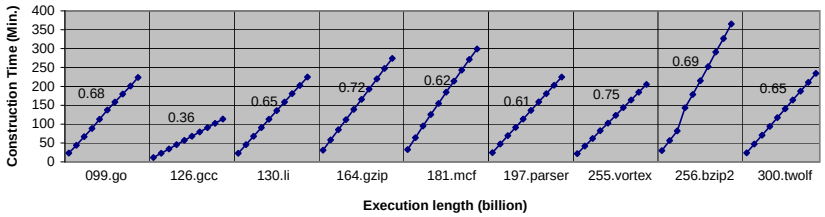


FIGURE 5.9. WET construction times.

5.4 Summary

In this chapter, a two tier compression strategy for value profiles is introduced, it provides a 11X compression rate and the feature of forward and backward bidirectional traversibility. The second tier, which is a generic prediction based compression technique, can be combined with the previously described *dynamic dependence graph* (DDG) optimizations to achieve 130X compression on DDG sizes. Dynamic slices can be computed for executions ranging from 114-130 millions intermediate statements on the compressed DDGs within 90 seconds.

Overall, the proposed two tier compression strategy can reduce the space consumption of a WET by the factor of 41.33. The execution of 3.9 Billion Trimaran intermediate code statements produces a compressed WET of 2 Gigabytes. In other words, the space efficiency is around 4 bits per statement.

Chapter 6
Pruning Backward Dynamic Slices Using Value Profiles

In this chapter, a fine-grained pruning technique of backward dynamic slices based on value profiles will be developed. The key observation is that by carefully examining the value profile, many of the statements in a backward dynamic slice can be determined to be highly unlikely to contain a fault. In this chapter, backward dynamic slices refer to dynamic slices in the form of dynamic dependence subgraphs instead of sets of statements. For ease of presentation, a different formulation of dynamic dependence graphs is used, in which each node corresponds to a single execution instance of a statement and each edge to a single exercising of a dependence. As a result, there is no need to refer to timestamps in order to distinguish between execution instances of a statement or a dependence. Note that the presentation of this technique does not distinguish between statement and statement instance. In other words, a statement refers to a statement instance if it is not otherwise specified.

6.1 Pruning Backward Dynamic Slices

This section gives an overview of the pruning technique. Given an observed incorrect value $\times_o$, a *Pruned Backward Dynamic Slice* of $\times_o$, $PDS(\times_o)$, can be computed, which contains a subset of statements from the *Backward Dynamic Slice* of $\times_o$, $DS(\times_o)$, that are likely to include faulty statements. Note that only the first incorrect output is considered because the backward dynamic slice of this output is usually the smallest among all the backward dynamic slices of incorrect outputs. Therefore, it is very likely to produce a small fault candidate set by pruning such a

slice.

It is observed that although $DS(\times_o)$ contains all statement instances that are involved in computing $\times_o$, not all of these statements are equally likely to be involved in causing the erroneous behavior. In particular, let us consider a common situation in which the program produces some correct outputs ($\surd_o$'s) before producing the incorrect value $\times_o$. From the perspective of the $\surd_o$'s and $\times_o$, it is possible to divide the executed statements in $DS(\times_o)$ into two sets: *May Set*, $DS_{may}(\times_o)$, containing executed statements from $DS(\times_o)$ that are also involved in computing one or more of the $\surd_o$ values; and *Must Set*, $DS_{must}(\times_o)$, containing executed statements from $DS(\times_o)$ that were involved in computing none of the $\surd_o$ values. In other words:

$$DS(\times_o) = DS_{must}(\times_o) \cup DS_{may}(\times_o)$$

$$DS_{must}(\times_o) = DS(\times_o) - \bigcup_{\surd'_o s} DS(\surd_o)$$

$$DS_{may}(\times_o) = DS(\times_o) - DS_{must}(\times_o)$$

While the statements in the $DS_{must}(\times_o)$ are always included in the pruned slice $PDS(\times_o)$, the ones in $DS_{may}(\times_o)$ may or may not be included in $PDS(\times_o)$. An analysis is developed that computes for value v computed by each statement execution $s \in DS_{may}(\times_o)$ a *confidence estimate* $C(v@s)$ between 1 and 0. High confidence estimate for a statement execution indicates that it is highly likely that the statement produced a correct value. Note that for simplicity it is assumed that one executed statement defines only one value. one The confidence estimates are computed using the *value profiles* of the executed statements. A *threshold confidence* τ is set such that only statement executions in $DS_{may}(\times_o)$ that have a confidence of less than τ are included in the pruned dynamic slice $PDS_\tau(\times_o)$. In other words:

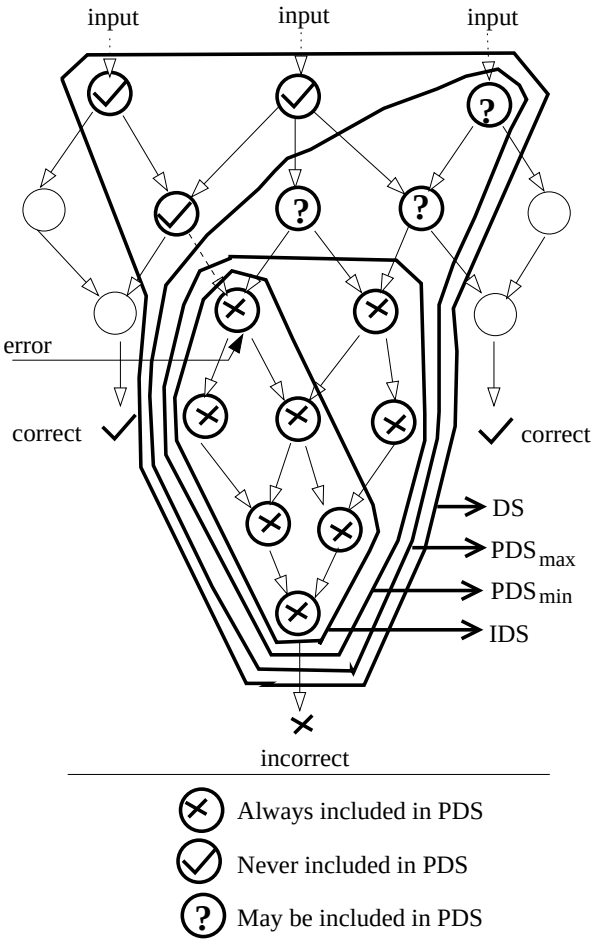


FIGURE 6.1. Pruning dynamic slice.

$$PDS_\tau(\times_o) = DS_{must}(\times_o) \cup DS^\tau_{may}(\times_o)$$

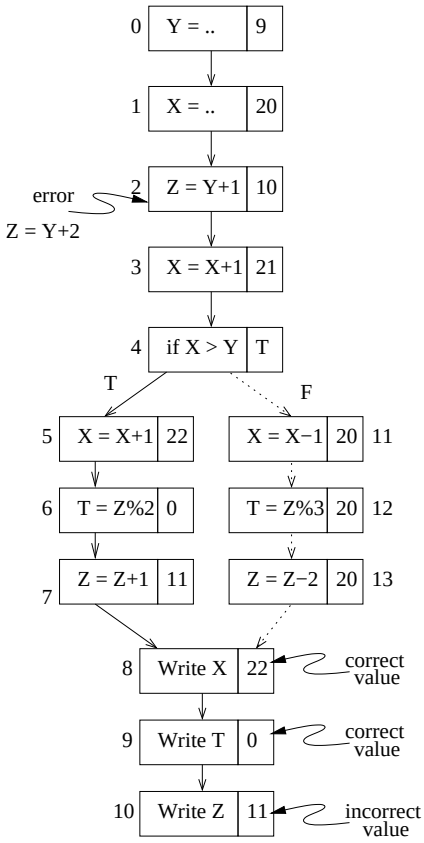
$$where,\ DS^\tau_{may}(\times_o) = \{s\ st\ s \in DS_{may} \wedge C(v@s) < \tau\}$$

It will be shown later that the analysis may yield confidence values of 1 for some statement executions and thus they are pruned from the dynamic slice irrespective of the choice of τ, i.e. they are never included in $PDS_\tau(\times_o)$ for all τ.

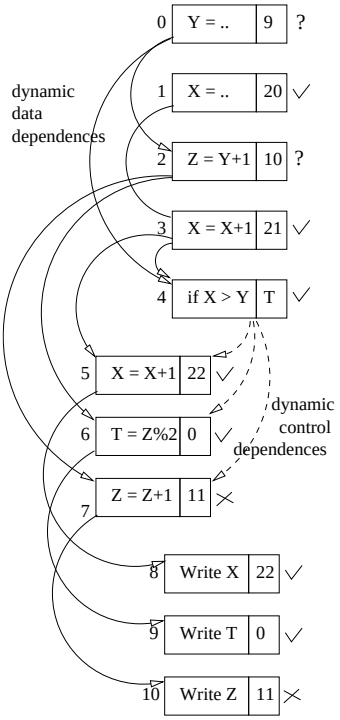
Figure 6.1 illustrates pruning of dynamic slices visually. It shows a dynamic dependence graph of a computation that produces two $\surd_o$ values before producing the incorrect value $\times_o$. *DS* is the *Dynamic Slice* of $\times_o$. Subset of nodes in *DS*

that form the *Ideal Dynamic Slice* (IDS) is shown – IDS originates at the point of program error and contains only those statement executions that produce erroneous values. The nodes in DS that are not present in IDS have been divided into three categories. The nodes labeled $\times$ in DS belong to DS_{must}, as they are not involved in computing the $\surd_o$ values, and thus they are always included in PDS the *Pruned Dynamic Slice* of $\times_o$. The remaining nodes in DS that are labeled with either $\surd$ or ?. The $\surd$ nodes have confidence value of 1 and thus they are never included in PDS. The nodes labeled ? have a confidence value of less than 1 and thus the value of threshold τ determines whether or not they are included in PDS. The identification of $\surd$ nodes is made possible by recognizing that *any change in the values produced by such nodes would alter the output values that were known to be correct.* Therefore it is assumed that these nodes must have produced correct values. As the figure shows, the smallest (largest) pruned dynamic slice that is produced by our algorithm corresponds to PDS_{min} (PDS_{max}). The key point to note here is that even if τ is set to 1, a pruned dynamic slice PDS_{max} is obtained, which is smaller than the dynamic slice DS. Note that PDS_{min} is actually what is known as a *dynamic dice* [20] – as the experiments later in this section show, often when faulty code is not captured by the dynamic dice it is captured by PDS_{max}.

Next a motivating example will be presented, which shows how analysis of code and runtime information can be used such that the confidence values of some statement executions in DS_{may} is determined to be 1. Figure 6.2(a) shows an execution of a program that follows the path corresponding to the true evaluation of the predicate at node 4. The value shown to the right of each statement is the value computed by the statement instance during execution. The dynamic dependence graph of this execution is shown in Figure 6.2(b) – the solid edges are data dependence edges while dotted edges are control dependence edges. The nodes in the dynamic slice of the incorrect output value produced by statement 10 include $\{0, 1, 2, 3, 4, 7, 10\}$. Now let us see how the correct outputs produced by statements 8 and 9 are used to mark the



(a) Program Execution.



(b) Dynamic Dependence Graph.

FIGURE 6.2. Pruning dynamic slice.

nodes in DS_{may} as $\surd$ or ?.

- From the correct output value of X written by statement 8 it is inferred that the values produced by statements 1, 3 and 5 are also correct. The reasoning on which this inference is based is as follows. The statements 3 and 5 represent *one-to-one mappings* between the used operand values and generated result values of X. Therefore any change in the values produced by statements 1, 3 or 5 will cause the value of output at statement 8 to change. However, the value of output at statement 8 is known to be correct. Thus, statements 1, 3 and 5 are marked with $\surd$ indicating that they produce correct values. It is further concluded that the *true* evaluation of predicate $X > Y$ is also correct. This is because if $X > Y$ would have evaluated to false, it would have produced a different output value for X at statement 8.

- Now let us consider the other correct output value written by statement 9. Since statement 6 does not represent a one-to-one mapping between its operand and result, even though the value of T that is produced by statement 6 is correct, one can not assume that the value of operand Z used in statement 6 is correct. As a result it is concluded that values produced by statements 0 and 2 may or may not be correct and therefore they are marked with a ?. Note that value of Y generated by statement 0 has another use in the predicate $X > Y$. Even though it has been determined that the predicate correctly evaluated to true, it cannot be determined from this fact that the value of Y used by the predicate is correct because many different values of Y would have produced the correct *true* evaluation of the predicate. Thus, from both uses of Y the same thing can be concluded, i.e. the value of Y produced by statement 0 may or may not be correct.

Given the above observations, the pruned dynamic slice of incorrect value output at statement 10 will always include statements 7 and 10. More importantly it will never

include statements 1, 3, 4 and 5. However, it may or may not include statements 0 and 2. The confidence values for statements 0 and 2 will be compared with the threshold τ to make this determination. In other words:

$$DS = \{0, 1, 2, 3, 4, 7, 10\}; \quad IDS = \{2, 7, 10\}$$

$$PDS_{max} = \{0, 2, 7, 10\}; \quad PDS_{min} = \{7, 10\}$$

In the remainder of this chapter a confidence estimation method will be discussed, which will produce the following results. First for the above example it will produce a confidence value of 1 for values produced by statements 1, 3, 4, and 5. Therefore the pruning algorithm will correctly remove statements 1, 3 and 4 from the dynamic slice of the incorrect output value produced by statement 10. Second it will produce confidence values of less than 1 for statements 0 and 2 such that the confidence value of statement 0 is more than confidence value of 2. Thus, depending upon the value of τ, three possible pruned slices will result: $\{0, 2, 7, 10\}$, $\{2, 7, 10\}$ and $\{7, 10\}$. The computation of confidence values will be performed using the *value profiles* of the executed statements (i.e., the operand values use and result values produced during statement executions).

6.2 Confidence Analysis

In this section an analysis will be developed, which will serve as the basis for pruning a conventional dynamic slice. The goal is to develop a *heuristic* for pruning a dynamic slice such that the size of the dynamic slice is significantly reduced and very rarely is the erroneous statement mistakenly pruned from the dynamic slice. In other words the objective is to significantly reduce the size of the slice with minimal loss in fault location effectiveness. As mentioned earlier, for ease of presentation, a different formulation of dynamic dependence graphs is used in which each node denotes a single execution instance and each edge denotes a single exercised dependence. The definition is shown as below:

Definition 2. *The **Dynamic Dependence Graph** of a program run, $DDG(N, E)$, consists of a set of nodes N and set of directed edges E where: each node $n_i \in N$ corresponds to i^{th} execution instance of statement n in the program; and each edge $m_j \rightarrow n_i \in E$ corresponds to a dynamic data dependence or dynamic control dependence of i^{th} execution instance of statement n on the j^{th} execution instance of statement m.*

In other words, with the execution of each statement during a program run, a new node is added to the dynamic data dependence graph and incoming edges to the node from other nodes on which the new node is data and control dependent are introduced.

The execution of every statement during a program run results in the computation of a result value. For an assignment statement this is the value assigned to the left hand side variable during the execution while for a predicate statement the value is either true or false corresponding to the result of predicate's evaluation. The dynamic slice of a value computed by a statement is defined as follows.

Definition 3. *Given $DDG(N, E)$, a dynamic dependence graph, the **Dynamic Slice** of $n_i \in N$ denoted by $DS(n_i)$ is the subgraph of $DDG(N, E)$ which includes n_i as well as all other nodes and edges from which n_i is reachable, i.e.*

$$DS(n_i) = (\{n_i\}, \{e | e = m_j \rightarrow n_i \in E\}) \cup \bigcup_{\forall m_j \rightarrow n_i} DS(m_j)$$

Consider a failed program run from which two kinds of evidence are collected: *negative evidence* in form of the first incorrect value $\times_o$ observed by the programmer during the program run; and *positive evidence* in form of some correct output values ($\surd_o$s) generated during the program run before the incorrect value $\times_o$ was generated. Each *relevant* value, i.e. value that was involved directly or indirectly in computing $\times_o$ and/or $\surd_o$ values, is classified into three distinct categories as defined below.

Definition 4. *A relevant value v generated by node n is classified as:*

- $\surd$ *or correct if it is used in computing at least one of the* $\surd_o$ *values but it is not used in computing the incorrect value* $\times_o$*. Therefore the values computed by all nodes in*
  $\bigcup_{\surd_o's} DS(\surd_o) - DS(\times_o)$ *are classified as* $\surd$*;*
- $\times$ *or incorrect if it is used in computing the incorrect value* $\times_o$ *but it is not used in computing any of the* $\surd_o$ *values. Therefore the values computed by all nodes in*
  $DS(\times_o) - \bigcup_{\surd_o's} DS(\surd_o)$ *are classified as* $\times$*; and*
- ? *or unknown if it is used in computing the incorrect value* $\times_o$ *and at least one of the* $\surd_o$ *values. Therefore the values computed by all nodes in* $DS(\times_o) \cap \bigcup_{\surd_o's} DS(\surd_o)$ *are classified as* ?*.*

As shown by earlier studies, dynamic slice $DS(\times_o)$ typically contains the erroneous code responsible for producing the incorrect value $\times_o$; however, it also includes many statement executions that are not responsible for generating the incorrect value. According to the above definitions, statement executions in $DS(\times_o)$ will be initially classified into two categories – some will be classified as $\times$ while others will be classified as ?. The ones that are classified as $\times$ are always included in the dynamic slice. However, the analysis is performed to determine what subset of statement executions classified as ? should be included in the pruned dynamic slice.

The decision as to whether the statement executions in the dynamic slice that are classified as ? should be included in the pruned dynamic slice is based upon *confidence analysis*. For every value v computed by statement execution n, confidence analysis produces a confidence estimate $C(v@n)$ that measures the likelihood of the value being correct. The confidence estimate $C(v@n)$ ranges from 0 to 1 where $C(v@n) = 0$ indicates that one has no confidence at all in the correctness of value

v while $C(v@n) = 1$ indicates that one has the highest possible confidence in the correctness of value $v@n$. This estimate is defined as shown below.

Definition 5. *Confidence estimate of value v computed by a relevant node n is defined as follows:*

- *if v is classified as $\surd$ (i.e., correct) then*

$$C(v@n) = 1$$

- *elseif v is classified as $\times$ (i.e., incorrect) then*

$$C(v@n) = 0$$

- *elseif v is classified as ? (i.e., unknown) then*

$$C(v@n) = 1 - log_{|Range(v@n)|}|Alt(v@n)|$$

where $Range(v@n)$ represents all legal values of v and $Alt(v@n) \subseteq Range(v@n)$ is a set of alternate values of v such that if any value in $Alt(v@n)$ was produced by n, the same correct $\surd_o$ values would have resulted.

Let us discuss the reasoning behind the $C(v@n)$ computation when v is classified as ?. If any change whatsoever in the value computed by n would cause at least one of the $\surd_o$ values to change and hence become incorrect, then it is concluded that the value v computed by n during the program run must have been correct. In this case the set $Alt(v@n)$ contains only one value. Therefore as desired, the confidence estimate $C(v@n) = 1 - log_{|Range(v@n)|}1 = 1$. On the other hand, if changing v to other values can still yield the same $\surd_o$ values, then we have less confidence in the correctness of value v. As the set $Alt(v@n)$ increases in size, the confidence estimate $C(v@n)$ reduces and when $Alt(v@n)$ is equal to $Range(v@n)$, then $C(v@n) = 1 - log_{|Range(v@n)|}|Alt(v@n)| = 0$.

Before settling on the above definition of confidence, other simpler definitions of confidence were also considered but they were found not be nearly as effective. For example, a definition has been considered in which each value's confidence was proportional to the number of correct outputs whose computation depended upon that the value. However, it was observed that in many cases different outputs were derived from different values and thus many values were assigned the same confidence. In addition, this simpler method fails to exploit the knowledge that sometimes even though a value may be involved in computing a single correct output, by looking at the statements involved it may be possible to definitely determine that the value is correct. For example, in Figure 6.2(b), since the value of X output by statement 8 is correct, it can be determined that the value of X computed by statement 1 must be correct. This is because the statements along the data dependence chain (4 and 5) perform one-to-one mapping between old and new values of X.

Next an algorithm is developed for computing confidence estimates. While othe definition of confidence estimates is quite simple, computation of confidence estimates is made challenging by the need for deriving the $Range(v@n)$ and $Alt(v@n)$ sets for all nodes n that are classified as ?. There are two key problems that must be addressed. First, given a variable x referenced by a program statement s, the set $Range(v@n)$ is first defined, which is the set of *legal values* that x may be allowed to take during its reference by an execution of s. Once such a legal set of values is determined for all variable references, the $Alt()$ sets will be computed with respect to these legal values. Second, an algorithm must be developed for propagation of values. Starting from the values classified as $\surd$, the dynamic dependence graph is traversed in a bottom up fashion to compute the $Alt()$ sets of values classified as ?. The $Alt()$ set of a value classified as $\surd$ is initialized to the singleton set containing the value while the $Alt()$ set of a value classified as ? is computed by examining the $Alt()$ sets of its child nodes in the dynamic dependence graph.

Let us first discuss how the set of *legal values* is determined for each variable

S: Y = ..
if (..)
C^1: X=Y+1
fi
if (..)
C^2: X=Y%2
fi
if (..)
C^3: X=Y+Z
fi

Reference	Value Profile
$Y@S$	{1,2,3,4,5,6,7,8,9}
$Y@C^1$	{7,8,9}
$X@C^1$	{8,9,10}
$Y@C^2$	{1,2,3,9}
$X@C^2$	{1,0,1,1}
$Z@C^3$	{9,9,5,5}
$Y@C^3$	{4,5,6,9}
$X@C^3$	{13,14,11,14}

FIGURE 6.3. Value profiles.

reference. A simple approach would be to use all possible values a variable can take based upon its type (integer, char, boolean) or compute a more accurate set using static analysis (e.g., range propagation [14]). However, such an overestimate is not very desirable for debugging because during debugging programmers are usually dealing with a single program execution (i.e. the failed run) corresponding to a specific program input. Therefore the *value profile* for the failed run is used to supply the set of legal values $Range()$.

Definition 6. *Given a reference (definition or use) to a variable v in a program statement s, the value profile $VP(v@s)$ provides an ordered list of values taken by variable v during the multiple executions of s in the failed run.*

Essentially, $VP(v@s)$ is equivalent to the val_s stream of the dynamic label sequence $[< t_s, val_s >]$ of node s. A program run generates a large number of values and exercises a large number of dynamic dependences. Capturing this history to perform dynamic slicing is a challenge already addressed by the techniques introduced in the previous chapters.

Now lets consider the rules of propagation along data and control dependence edges in the dynamic dependence graph. Intuitively, given an execution instance of a statement, the values in the $Alt()$ set of the result computed by the statement are constrained by each of its children in the dynamic data dependence graph. Only

those values can be put into the $Alt()$ set that do not adversely impact any of the $\surd_o$ values along any chain of dependence edges from the executed statement to any of the $\surd_o$ values. Therefore, $Alt()$ sets are also associated with dynamic dependence edges and then the $Alt()$ set for result value of an executed statement is simply computed by intersecting the $Alt()$ sets of edges leaving the statement. There are two key operations involved in propagation. First from $Alt()$ set of a result computed by an executed statement, the subset of legal values that the operands can take is computed such that these operand values produce result values contained in the $Alt()$ set. Second the $Alt()$ set of a result is computed by examining the subset of legal values already determined at each of the uses of the result value.

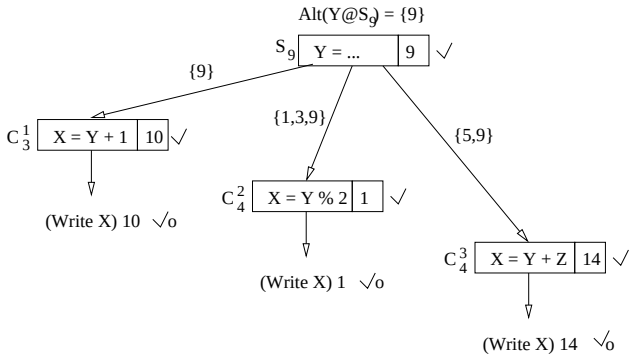


FIGURE 6.4. Dependences among assignment statements.

Let us consider propagation along dynamic data dependence edges that connect assignment statements (we will also consider predicate statements shortly). We illustrate propagation by analyzing the result value computed by 9^{th} execution instance of statement S in the example from Figure 6.3. The value of Y computed by S_9 is 9 and this value is used later by the 3^{rd} instance of statement C^1, the 4^{th} instance of statement C^2, and the 4^{th} instance of statement C^3. The dynamic dependence will therefore include three data dependence edges $S_9 \rightarrow C_3^1$, $S_9 \rightarrow C_4^2$, $S_9 \rightarrow C_4^3$. We further assume that the values of X computed by C_3^1, C_4^2 and C_4^3 are output and determined to be correct. Figure 6.4 first shows how the potential values in $Alt(Y@S_9)$ set are identified by considering each dynamic data dependence individually. Given

that C^1 represents a *one-to-one mapping* between the value operand Y and result X (determined from value profiles), the $Alt()$ set assignment of Y at S_9 constrained by C_3^1, denoted by $Alt(Y@S_9 \rightarrow C_3^1)$, contains 9. In contrast, since statements C^2 and C^3 do not represent *one-to-one mappings* between the value of operand Y and the value of result X, the sets $Alt(Y@S_9 \rightarrow C_4^2)$ and $Alt(Y@S_9 \rightarrow C_4^3)$ corresponding to dynamic data dependence edges $S_9 \rightarrow C_4^2$ and $S_9 \rightarrow C_4^3$ contain more than one value. However, the $Alt(Y@S_9)$ is computed by intersecting the three sets for the three dynamic data dependences yielding a set with only one element. Therefore the confidence estimate $C(Y@S_9) = 1$ and therefore the value computed by S_9 is marked as $\surd$, i.e. correct.

From the above analysis two things can be observed. First, the presence of one-to-one mappings is greatly beneficial in pruning a dynamic slice since they prevent $Alt()$ sets from expanding as propagation proceeds. Second, it is observed that as long as their is one data dependence edge along which a computed can be verified (i.e., its $Alt()$ set contains one value), the value is considered verified. It will be shown later that the approach is very effective because programs often contain many statement executions that correspond to one-to-one mappings (e.g., copy operations, expressions with two operands one of which is a constant etc.).

In the above example the propagation along dynamic data dependence edges is considered and these edges were present between assignment statement executions. Next it will be discussed how to handle the situation in which predicate evaluations are present and hence dynamic control dependence edges are also present. There are two points to be made here. First the value of a predicate is classified as being correct ($\surd$) if the value of one of its direct or indirect control dependent assignment statements has been determined to be $\surd$. This is because if the predicate would have evaluated differently the variable assigned by the control dependent assignment would have had a different value and hence it would have adversely affected on of the $\surd_o$ values through its further uses. Second it should be noted that when the result value

of a predicate is classified as correct, it only means that the outcome of the predicate evaluation (true or false) is correct. However, since a predicate usually represents a *many-to-one* mapping between its operand values and true/false result, it cannot be inferred that the operand values are necessarily correct. The only thing one can say is that the operand values are the subset of legal values for which the predicate produces the same desired result, i.e. true/false. To illustrate the above points a fragment of the previous example is used as shown in Figure 6.5. The dynamic dependence graph and the results of analysis are shown in the figure. Note that the predicate evaluation P_9^1 is marked as $\surd$ because its dynamic control dependent child C_3^1 is marked $\surd$. $Alt(Y@S_9 \rightarrow P_9^1)$ also includes values 7 and 8 in addition to 9 as for these legal values of Y, the predicate $Y > 6$ evaluates to true just as it evaluates to true for the value 9 produced by S_9.

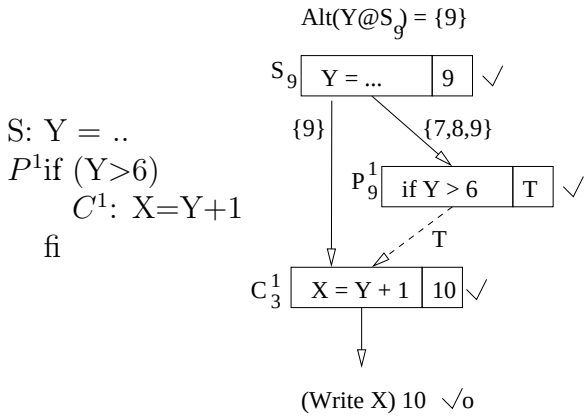


FIGURE 6.5. Dependences involving predicates.

The process described is summarized fully in the algorithm presented in Figure 6.6. All nodes in the dynamic dependence graph that have been marked as ? are the ones that are processed to compute their confidence estimates. The $Alt()$ sets for all nodes are initialized to the set containing the value $val()$ produced by the node. The nodes marked ? are then processed in a bottom-up order one by one. If a node being processed is an assignment statement then the $Alt()$ set for its result value is computed, from which then its confidence estimate is derived. Predicate nodes

are processed by considering the markings on their dynamically control dependent assignment statements. In Figure 6.6, the function $ComputeAlt()$ presents the details of the $Alt()$ set computations which were described intuitively earlier.

6.3 Experimental Results

TABLE 6.1. Characteristics of benchmarks

Benchmark	Version	Error in	Failed Cases	Position Range
print_tokens (565 LOC)	1	switch-case	6	[14-495]
	2	switch-case	143	[17-1707]
	4	constant	23	[17-1209]
	6	constant	143	[13-2714]
	7	predicate	28	[8-1271]
print_tokens2 (510 LOC)	4	assignment	268	[20-394]
	5	return	67	[20-1106]
	6	parameter	329	[20-870]
	7	predicate	158	[27-486]
	8	predicate	194	[60-928]
replace (563 LOC)	1	predicate	24	[2-20]
	3	predicate	130	[2-666]
	6	loop condition	92	[2-609]
	9	predicate	92	[2-609]
	14	predicate	92	[3-49]
	18	predicate	190	[2-380]
	21	predicate	2	[18-40]
2-5	25	predicate	2	[3-11]
schedule (412 LOC)	2	assignment	200	[2-38]
	4	predicate	267	[2-39]
	7	added code	20	[2-14]
schedule2 (307 LOC)	5	added code	32	[5-28]
	6	constant	2	[10-18]
	7	predicate	20	[2-16]
gzip (7199 LOC)	1	predicate	6	[19-19]
flex (12418 LOC)	4	constant	12	[16885-53109]
	5	constant	257	[7130-9056]
	7	constant	97	[6164-6164]
	10	array index	6	[7142-7144]
	11	predicate	513	[6867-43647]
	15	constant	515	[13430-53895]
	17	constant	315	[10632-51067]
	19	constant	343	[20495-61777]

6.3.1 Benchmarks used

Table 6.1 shows the benchmarks used in the experimentation. The first five are known as Siemens suite programs [43]. The last two unix utilities are also available from the

Initialize: $Alt(*) \leftarrow \{val(*)\}$;
for each relevant node S_i marked ? in *bottom-up order* do
  if S_i is an *assignment* $X = ..$ then
    $ComputeAlt(Alt(X@S_i))$;
    if $|Alt(X@S_i)| = 1$ then
      $C(X@S_i) = 1$; *mark* S_i as $\surd$;
    else
      $C(X@S_i) = 1 - log_{|Range(X@S_i)|}|Alt(X@S_i)|$
    endif
  elseif S_i is a *predicate* then
    if $\exists$ S_j st S_j dynamically control dependent upon S_i
        and S_j is marked $\surd$
    then *mark* S_i as $\surd$ endif
  endif
endfor

$ComputeAlt(Alt(X@S_i))$
  Let the following dynamic dependence edges lead
  from S_i to nodes marked $\surd$ or ?:
  to assignments: $S_i \rightarrow C^1_{1i}$, $S_i \rightarrow C^2_{2i}$, ... $S_i \rightarrow C^n_{ni}$;
  to predicates: $S_i \rightarrow P^1_{1i}$, $S_i \rightarrow P^2_{2i}$, ... $S_i \rightarrow P^m_{mi}$.

  for each $C^j : Y = f(X)$ st $\exists$ $S_i \rightarrow C^j_{ji}$ do
    $Alt(X@S_i \rightarrow C^j_{ji}) = \{v :$
      $v \in VP(X@C^j) \wedge C^j(X = v) \in Alt(Y@C^j_{ji})\}$
  endfor
  for each $P^j : f(X)$ st $\exists$ $S_i \rightarrow P^j_{ji}$ do
    $Alt(X@S_i \rightarrow P^j_{ji}) = \{v :$
      $v \in VP(X@P^j) \wedge P^j(X = v) = P^j_{ji}\}$
  endfor
  $$Alt(X@S_i) = \bigcap_{\forall j, S_i \rightarrow C^j_{ji}} Alt(X@S_i \rightarrow C^j_{ji}) \cap \bigcap_{\forall j, S_i \rightarrow P^j_{ji}} Alt(X@S_i \rightarrow P^j_{ji})$$
end$ComputeAlt$

FIGURE 6.6. Confidence computation algorithm.

same website [2]. This suite of programs was used because it provides several faulty versions of the programs which have exactly one fault injected in them. The versions used in the experiments are also indicated in Table 6.1. For each faulty version many test inputs are provided [43]. Different inputs result in different position for the first incorrect output in the output stream. The column *position range* of Table 6.1 gives the range of the position of the first observed wrong output. The greater is the position number, the greater is the number of correct outputs produced before the incorrect output. One can see that it is common for a certain number of correct outputs to be generated in a failed run. In fact these numbers can be very high for some inputs.

The test suite provides more versions than those used in the experiments. some of the versions were excluded as they are not appropriate for experimentation. Some versions produced no output or the very first output produced was wrong. Therefore our approach was not applicable. In two kinds of situations the faulty statement was not present in the dynamic slice itself and thus the effectiveness of pruning could not be studied in such cases. First, *code omission* faults were present in some versions. Since such faults were not even captured in the static slice of the output, they could not be caught any by any dynamic slicing algorithm. Second, it was mentioned earlier that a dynamic slice does not always include the erroneous executed statement. This happens when the erroneous output is produced due to an incorrect evaluation of a branch predicate causing execution of some statements to be incorrectly bypassed. This situation can be handled by *relevant slicing* [33, 88]. While in our experiments such cases were omitted, later it will be shown how they can be handled by augmenting the technique.

6.3.2 Confidence-based Pruning

Since for some faulty versions there are many test inputs, and some of these may not differ much in their behavior, for each faulty version three test inputs were selected such that varying number of correct outputs are generated before the incorrect output is produced. Whenever possible, three runs were selected such that the wrong output was observed at: the lower bound of *position range* in the first run; closest to the middle of *position range* in the second run; and at the upper bound of *position range* in the third run. For each run, the dynamic slice of the wrong output was first computed and then the slice was pruned using confidence analysis. Six numbers are presented about the slice sizes in Tables 6.2 and 6.3. $All.PDS_{min}$, $All.PDS_{max}$, and $All.DS$ represent the number of DDG nodes in PDS_{min}, PDS_{max}, and DS. The corresponding *distinct* numbers ($D.PDS_{min}$, $D.PDS_{max}$, and $D.DS$) denote the number of unique statements in them (note that one unique statement may get executed many times and result in many nodes in DDG). We also present the fault location effectiveness in column *Error In*. Here I, X, and D indicate the presence of erroneous statement in PDS_{min}, PDS_{max}, and DS respectively. The results are also summarized by taking averages across different versions of each benchmark in Table 6.4.

From these tables, the following observations can be made:

1. The confidence analysis greatly reduces the size of dynamic slice without sacrificing the fault location effectiveness. Table 6.4 shows the average factor by which PDS_{max} is smaller than DS ranges from 4.31 to 87514.33 (all) and 1.79 to 26.93 (distinct). For *flex*, the slices are so precisely reduced that they simply contain the chain of dependences from the erroneous statement to the incorrect output – this chain includes only a few statements.

2. For most of the versions, three runs were used and the relation between the

Benchmark	Version	Wrong Output Pos.	($All.PDS_{min}$ -$All.PDS_{max}$)/$All.DS$	($D.PDS_{min}$ $D.PDS_{max}$)/$D.DS$	Error In
print_tokens	1	14	(310-310)/712	(41-41)/72	IXD
		301	(239-240)/4582	(40-40)/86	IXD
		495	(317-317)/13603	(41-41)/134	IXD
	2	17	(70-70)/429	(19-19)/61	IXD
		231	(68-69)/3605	(18-18)/86	IXD
		1707	(70-70)/44158	(19-19)/149	IXD
	4	17	(246-246)/603	(40-40)/69	IXD
		91	(212-212)/1965	(35-35)/92	IXD
		1206	(263-295)/28513	(43-43)/141	IXD
	6	13	(1457-1470)/1804	(44-44)/71	IXD
		109	(214-214)/1993	(35-35)/97	IXD
		2714	(432-432)/66651	(36-36)/145	IXD
	7	$8^{(1)}$	(399-400)/698	(41-41)/74	IXD
		$92^{(1)}$	(423-436)/1486	(41-41)/94	IXD
		$1271^{(1)}$	(390-391)/27274	(37-37)/136	IXD
print_tokens2	4	20	(174-174)/902	(40-40)/99	IXD
		47	(447-447)/1561	(50-50)/95	IXD
		394	(770-770)/8364	(44-44)/138	IXD
	5	20	(499-499)/850	(58-58)/97	IXD
		79	(364-364)/1013	(59-59)/109	IXD
		1106	(285-285)/27841	(56-56)/154	IXD
	6	20	(208-208)/680	(61-61)/95	IXD
		34	(208-208)/770	(61-61)/97	IXD
		870	(208-208)/18602	(61-61)/143	IXD
	7	27	(697-698)/1290	(59-60)/96	IXD
		75	(329-329)/1140	(53-53)/83	IXD
		486	(1105-1105)/10630	(67-67)/148	IXD
	8	$60^{(1)}$	(377-377)/2091	(59-59)/100	IXD
		63	(377-406)/1676	(48-51)/105	IXD
		928	(367-413)/20738	(48-51)/151	IXD
replace	1	2	(192-494)/2212	(38-77)/147	XD
		9	(241-461)/1625	(53-81)/130	XD
		20	(179-408)/1687	(44-64)/128	XD
	3	2	(160-671)/1012	(32-86)/136	IXD
		18	(89-89)/1997	(21-21)/155	IXD
		666	(17-868)/18522	(3-45)/125	XD
	6	2	(371-780)/1166	(45-62)/136	IXD
		19	(216-648)/2129	(28-50)/132	IXD
		609	(325-605)/20525	(46-49)/153	IXD
	9	2	(180-357)/889	(40-61)/115	XD
		$26^{(2)}$	(48-243)/3047	(18-42)/125	D
	14	3	(289-656)/1187	(55-88)/138	IXD
		9	(1006-1689)/2515	(73-117)/161	IXD
		49	(103-112)/3021	(23-28)/111	IXD
	18	2	(106-107)/669	(26-27)/109	IXD
		35	(152-152)/4145	(37-37)/143	IXD
		380	(194-194)/12588	(37-37)/127	IXD
	21	18	(390-781)/2372	(53-86)/132	XD
		40	(502-783)/3501	(42-59)/102	XD
	25	3	(321-531)/975	(55-78)/120	IXD
		11	(450-552)/2952	(72-84)/165	IXD

(1). Part of the wrong output appeared to be correct;
(2). The root cause was pruned.

TABLE 6.2. Pruning effectiveness results of faulty versions for up to three test inputs.

Benchmark	Version	Wrong Output Pos.	($All.PDS_{min}$ -$All.PDS_{max}$)/$All.DS$	($D.PDS_{min}$ $D.PDS_{max}$)/$D.DS$	Error In
schedule	2	2	(464-465)/1046	(65-66)/93	IXD
		10	(621-623)/2155	(69-69)/118	IXD
		38	(295-359)/6176	(55-55)/119	IXD
	4	2	(1225-1468)/2605	(88-98)/119	IXD
		10	(1025-1029)/2155	(85-89)/117	IXD
	7	2	(386-399)/726	(67-68)/90	IXD
		6	(83-284)/1124	(24-65)/105	XD
		14	(84-330)/2146	(24-59)/97	XD
schedule2	5	5	(1152-1152)/1823	(64-64)/83	IXD
		14	(195-195)/2594	(34-34)/73	IXD
		28	(1896-1896)/5639	(60-60)/79	IXD
	6	10	(230-230)/1611	(40-40)/67	IXD
		18	(254-254)/2526	(42-42)/67	IXD
	7	2	(80-145)/696	(27-36)/67	IXD
		6	(113-129)/2871	(25-27)/94	IXD
		16	(693-709)/3311	(59-61)/84	IXD
gzip	1	19	(82-394520)/1699490	(10-121)/357	XD
flex	4	16885(1)	(13-14)/62235	(7-8)/692	IXD
		19825(1)	(16-17)/42823	(9-9)/648	IXD
		53109(1)	(13-14)/1120244	(7-8)/889	IXD
	5	7130	(17-76)/23292	(6-18)/542	IXD
		8925	(4-4)/81991	(3-3)/681	IXD
		9056	(4-4)/59501	(3-3)/709	IXD
	7	6164	(17949-18026)/22886	(217-229)/280	IXD
	10	7142	(76-86)/84210	(19-23)/730	IXD
		8925(1)	(74-75)/1021249	(17-18)/786	IXD
	11	6867	(15-15)/5756	(10-10)/81	IXD
		16092	(15-15)/39484	(10-10)/552	IXD
		43647	(15-15)/254532	(10-10)/720	IXD
	15	13430(1)	(71-71)/30002	(14-14)/824	IXD
		16092(1)	(71-71)/72756	(14-14)/988	IXD
		53859(1)	(96-96)/1120987	(19-19)/941	IXD
	17	10632	(1-1)/22093	(1-1)/632	IXD
		11584	(1-1)/86515	(1-1)/813	IXD
		51067	(1-1)/1118733	(1-1)/864	IXD
	19	20495(1)	(35-54)/32219	(16-20)/764	IXD
		21955(1)	(35-35)/98133	(16-16)/947	IXD
		61777(1)	(32-33)/1130822	(15-16)/981	IXD

(1). Part of the wrong output appeared to be correct;

TABLE 6.3. Pruning effectiveness results of faulty versions for up to three test inputs.

Benchmark	$(All.PDS_{min} - All.PDS_{max})/$ $All.DS$	$(D.PDS_{min} - D.PDS_{max})$ $D.DS$	$All.DS/$ $All.PDS_{max}$	$D.DS/$ $D.PDS_{max}$
print_tokens	(341-345)/1320	(35-35)/100	73.4	3.12
print_tokens2	(428-433)/6543	(55-55)/114	19.53	2.09
replace	(310-546)/4112	(43-60)/131	13.14	2.52
schedule	(454-596)/3188	(56-70)/117	9.41	1.79
schedule2	(562-630)/2358	(50-58)90	6.58	1.69
gzip	(82-394520)/1699490	(10-121)/357	4.31	2.95
flex	(1232-1240)/342692	(25-27)727	276.36	26.93

Benchmark	$All.PDS_{max}/All.PDS_{min}$		$D.PDS_{max}/D.PDS_{min}$	
	IX	X	IX	X
print_tokens	1.01	NA	1	NA
print_tokens2	1.01	NA	1.01	NA
replace	1.78	8.55	1.38	3.36
schedule	1.08	3.68	1.03	2.58
schedule2	1.54	NA	1.29	NA
gzip	NA	4811.22	NA	12.1
flex	1.05	NA	1.04	NA

TABLE 6.4. Summary of results across all versions.

pruning capability and the number of correct outputs was studied. From Tables 6.2 and 6.3 it is observed that the absolute sizes of the PDSs appear to be independent of the number of correct outputs. However, the reductions in the sizes of PDSs with respect to the sizes of DSs increase as the number of correct outputs grow because of the increases in the sizes of DSs.

3. It is observed that the fault location effectiveness of PDS_{max} is very good. Even though it is much smaller than DS, only in one case the erroneous statement is removed during pruning – this happened in *replace* version v9 run r2. Figure 6.7 explains how this happened. In this run, statement $i = i + 1$ is wrong such that 'D' is assigned to the wrong position in array *pat*. However, statement *return flag* is verified and thus *flag=true;* is verified, which means the predicate is correct. Since the predicate represents a one-to-one mapping to its operand when it evaluates to *true*, $pat[j]$ contains the correct value 'D'. According to the analysis, the store to $pat[i]$ will get verified and so will the wrong index. As is illustrated in the right hand side of Figure 6.7, $pat[j]$ being correct is the result of both array *pat* and j being wrong. A plausible solution is not to

infer the correctness of j from the correctness of $pat[j]$. However, it becomes so conservative that the effectiveness of pruning diminishes.

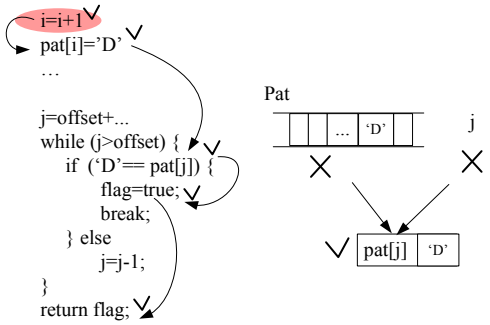


FIGURE 6.7. Replace v9 r2

4. Let us compare PDS_{max} with PDS_{min}. Although PDS_{min} works for a large number of test cases, it was observed that in several cases, such as *replace v1, v3, v9, v21* and *schedule v7*, it prunes the erroneous statement while PDS_{max} does not do so. On the other hand, PDS_{max} works almost equally well for the cases in which PDS_{min} also works. As shown in Table 6.4, when the erroneous statement is captured in both PDS_{max} and PDS_{min}, corresponding to the IX columns, PDS_{max}/PDS_{min} is roughly one, i.e. their sizes are nearly the same (the entries marked NA are ones where there were no slices in that category). Thus, using confidence analysis to obtain PDS_{max} is an effective method for both pruning the slice and maintaining the fault location effectiveness.

5. In some cases such as *flex v15*, part of the wrong output appears to be correct which may cause some confusion. For example, *flex v15* has the error of *printf ("YY_USER_ACTION")* missing a '$\backslash n$' at the end of the string. If we assume the "YY_USER_ACTION" is correct, the wrong *printf* will get verified. To solve this problem, the output is divided into units, which is lines in this case, and compute slice on the first character of the wrong unit.

6.3.3 Enhancements to Pruning

With the help of a programmer. It is possible that PDS_{max} is still quite big. However, pruning can be further carried out during debugging. During the course of debugging the programmer usually investigates the values in *gdb* and decides if they are correct or wrong. This information can be fed back to the confidence analysis to enable further pruning. Similarly, the programmer can also look at the slice and tell the system if certain values seem to be correct. An experiment was conducted trying to simulate this procedure. *Replace* version v14 was picked, one of whose three prunings (the third run) was quite successful and the relation from the error to the wrong output could be understood. The largest pruned slice in the third run, PDS^3_{max}, was used as a reference to examine the PDS^1_{max} of the first run. The first statement instance which was in PDS^1_{max} but not in PDS^3_{max} was found and marked as correct by the system. PDS^1_{max} was further pruned to 587/74 (*all*/*distinct*) from 656/88. After another two interactions, it was reduced to 93/23, which was very close to dependence chain along which error was propagated. The same experiment was also tried with *replace v3* – the second run was used as a reference to prune the first run and it was found that in only one step, the slice was reduced from 671/86 (*all*/*distinct*) to 33/15 and it still contained the error.

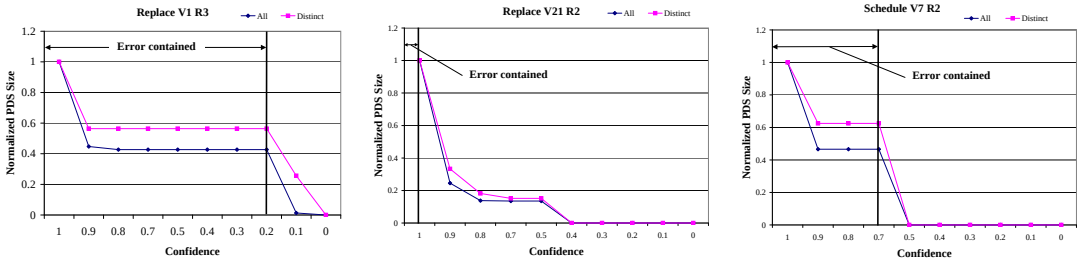


FIGURE 6.8. Pruned dynamic slice for varying threshold (version Vi run Rj).

Looking for a threshold. So far either PDS_{min} or PDS_{max} were discussed. In this experiment, the relationship between the threshold τ and the corresponding PDS_τ's size and its fault location effectiveness is studied. The results for three

different runs are plotted in Fig. 6.8. As expected, the PDS_τ drops in both size and fault location effectiveness as τ decreases. However, the existence of such a τ that nicely balances between the size and fault location effectiveness was not observed.

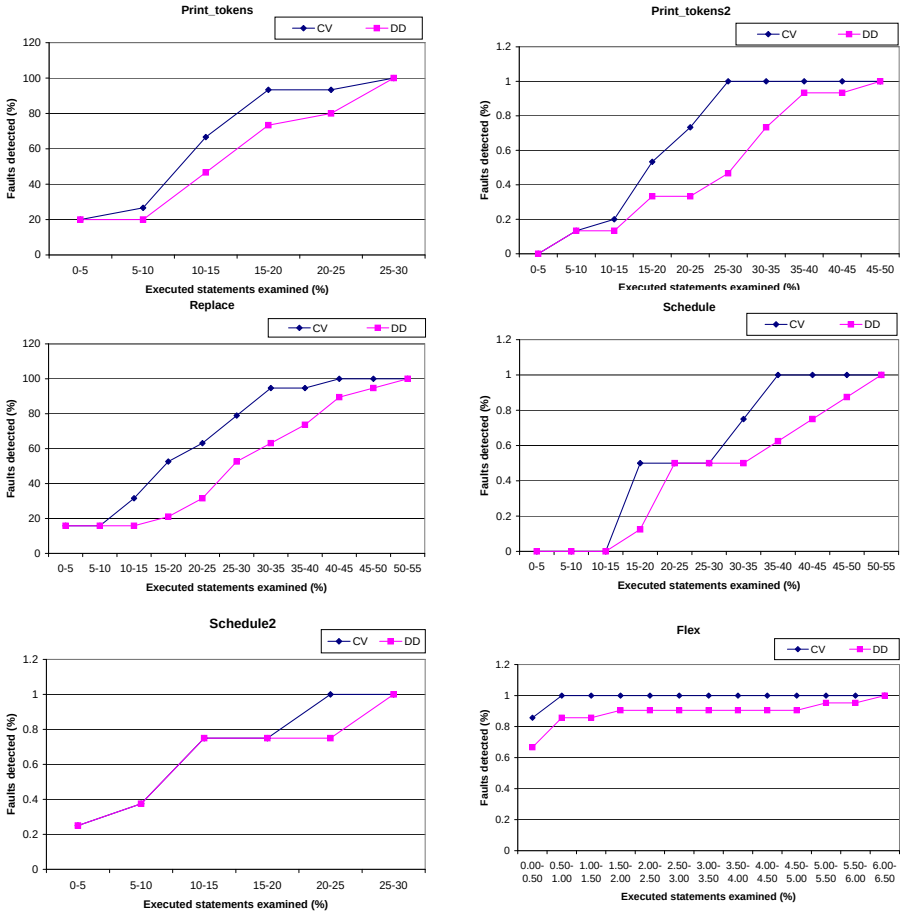


FIGURE 6.9. Locating fault by examining statements in increasing order of confidence values.

Prioritization based on confidence. As mentioned earlier, it was observed that the most effective pruning strategy is one in which only the statements with confidence values of 1 are pruned from the dynamic slice to produce PDS_{max}. Next an additional use of confidence values is studied. The statements in PDS_{max} are

prioritized in the order of increasing confidence values. To locate faulty code, the statements are then examined by the programmer in the order of increasing confidence values till the faulty code is encountered. The effectiveness of this strategy is measured in terms of the percentage of executed statements that are examined by the programmer before encountering the faulty code.

In prior work it was shown that an effective strategy for exploring dynamic slices to locate the faulty code is to examine the statements in the dynamic slice in increasing order of their dependence distance from the point at which the erroneous value is encountered during a failed run [88, 84]. An experiment was conducted in which the effectiveness of two strategies were compared: exploring dynamic slice in order of increasing *dependence distances* (DD); and exploring pruned dynamic slice PDS_{max} in the order of increasing *confidence values* (CV). When using the confidence value based strategy, if two statements with same confidence value are present, then the dependence distance is used as the tie-breaker.

The results of this experiment are given in Figure 6.9. For a given point in each graph, the y-axis represents the fraction of faults located while the x-axis represents the percentage of executed statements examined to locate these faults. The results are the averages over the three failed runs that were used in the experiments presented in the preceding section. As one can observe, for a given percentage of executed statements examined, typically the fraction of faults that are located is higher for CV in comparison to DD.

6.4 Summary

In this chapter, a novel approach for pruning dynamic slices using positive evidence was introduced, which exploits program state information in terms of observed values of variables in addition to the dynamic dependence information as is done traditionally in dynamic slicing. A simple analysis was developed to estimate a confidence

value for any computed value. Due to a fairly large number of executed statements that represent one-to-one mappings between an operand and the result, the highest confidence value of one is obtained for a large number of computed values. As a result, even the largest pruned dynamic slice computed is significantly smaller than the conventional dynamic slice. The number of distinct statements in PDS_{max} is 1.79 to 26.93 times less than the corresponding number in DS. Confidence analysis was not evaluated on the real bugs introduced earlier because most of those faulty programs do not produce any correct output. Therefore, even though it is strongly believed by the author that it is very common for a failing run to produce partially correct output in real life, further empirical studies should be carried out in future. More importantly, if both negative and positive evidences are collected in one single failing run, further reduction on the fault candidate set can be achieved. In the next chapter, it will be shown how even longer program runs can be handled by identifying relevant intervals of execution and then limiting slicing only to these intervals.

Chapter 7
Dynamic Slicing of Long Running Programs

The optimization and compression techniques discussed in earlier chapters achieve the space efficiency of 4 bits per instruction. A simple task as starting Mozilla and browsing a html page may create traces with the size of a few giga bytes. In other words, tracing based techniques, such as dynamic slicing, can handle executions up to a few seconds given the speed and storage capacity of today's workstations. Realistic executions with the lengths of minutes, hours, or days seem to be far beyond the capability of dynamic slicing given all the advances. This chapter discusses a plausible solution.

7.1 Overview

While a naive solution is to divide the entire execution by checkpoints, and then apply dynamic slicing enabled by tracing on one checkpoint interval at a time. However, this solution is not as simple as it appears for two reasons. First, tracing requires instrumenting the original program. There are two kinds of instrumentation techniques – static and dynamic. Static instrumentation, in which the program is instrumented by compilers, introduces non-trivial execution overhead as tracing cannot be easily turned off. Dynamic instrumentation adaptively instruments the program. It can easily switch from executing the original code to executing the instrumented code or vice versa. Dynamic instrumentation engine usually resides in the process's virtual space and manipulates the virtual memory intensively such that the status of the application process is substantially mixed with the instrumentation engine's status.

While checkpoints are often produced by taking snapshots of the virtual memory, it becomes hard to discretely checkpoint the application process. Second, tracing can handle executions up to a few seconds. In contrast, checkpointing usually produces virtual memory snapshots with the size of a few mega bytes, it is not something that can be easily afforded to perform every second. Checkpoints are usually created in the interval of, more or less, minutes. The gap between seconds and minutes suggests that it is still too costly to trace a checkpoint interval.

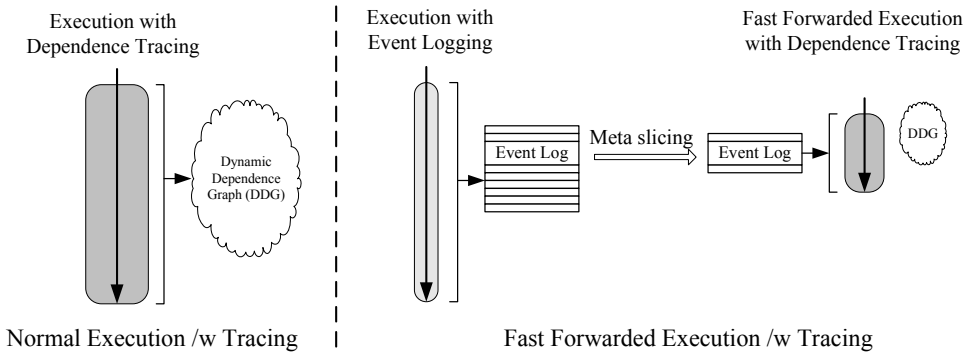


FIGURE 7.1. Execution fast forwarding.

Figure 7.1 gives an overview of the idea. The left part illustrates that an execution, or part of an execution delimited by checkpoints, is usually heavily instrumented for the purpose of dependence tracing. The heavy instrumentation introduces very high runtime overhead and constructs a huge dependence graph, which makes it impractical if the execution gets long. In the right part an *execution fast forwarding* technique takes advantage of the characteristics of many long running programs – being driven by events. More precisely, it first collects a full event log from the original execution; given a specific part of the execution that the programmer wants to replay, a meta slicing technique, which is analogous to dynamic slicing but performed on logged events instead of executed instructions, is applied to prune the events irrelevant to replaying the desired execution region. The reduced event log is used to drive the replay. Compared to the original run, the *fast forwarded* execution is much smaller

as the volume of events passed to the program is much lower. As a result, a smaller dependence graph is generated that can be collected through tracing.

7.2 Execution Fast Forwarding

Often when a program runs for a long time it is not because it performs a very long and complicated task. Instead, it is often because the program processes a long sequence of simple tasks. For example, programs processing streaming data such as audio, video, and packet data usually carry out the same computation e.g. FFT transformation on a sequence of data; the computation on each data piece tends to be relatively lightweight and independent from each other. Programs that require user interactions display similar properties: the programs spend most of their execution time in handling user actions and the computation dedicated for each user action is usually simple. Server programs deal with thousands of requests, most of which set off simple computations such as reading a file or retrieving a piece of data from a database. A common feature of these programs is that *they are driven by events*. The events divide the whole execution into small tasks, each one of which corresponds to handling some event. An event is defined as one interaction between the application and the OS. The interaction could be in the forms of: system calls such as *open, read*, and *mmap2*; asynchronous or synchronous signals such as *kill* and *segfault*. These events are used to provide OS services, for instance reading/writing a file/socket, to the application program, or to notify something has happened.

An *execution fast forwarding* (EFF) technique is derived from the following observation – *all the events do not need to be replayed in order to replay a particular part of the execution*. Given the fact that the execution is driven by events, we may be able to shrink the replayed execution, and yet reproduce the desired part, if the irrelevant events can be pruned.

Figure 7.2 presents a motivation example. In the original run, the key '*c*' was

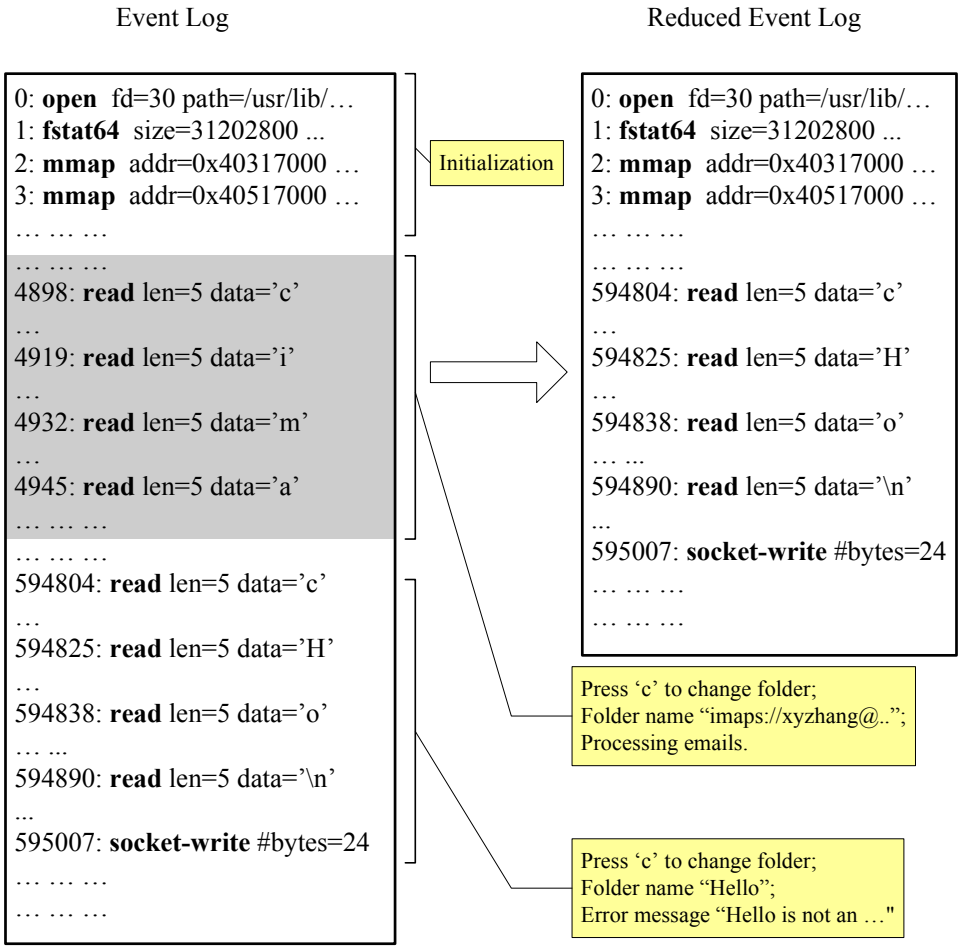


FIGURE 7.2. Getting the same warning message by replaying the reduced log for Mutt 1.4.2.1i. The numbers mean the byte positions of the corresponding events in the log.

first pressed in order to change the folder name after Mutt, a text based mail user agent, was started; string "*imaps://xyz- hang@email.cs.arizona.edu/inbox*" was typed in as the email account, which was followed by the password; after logging in the account, a couple of email messages were accessed; then '*c*' was typed again and string "*Hello*" was provided as the new folder name. Since "*Hello*" was not a valid folder name, a warning message was printed on the screen. The events were logged in a file as shown on the left hand side of the figure. The first a few thousands of events present the startup phase of the execution, which is mainly about loading

dynamic libraries, allocating virtual memory, and initializing the program state. The shaded events starting from byte position 4898 to position 594803 correspond to the execution related to accessing the email account. Events starting from 594804 contribute to entering the invalid folder name and the warning message was printed by the event at 595007. Let us assume the programmer is interested in reproducing the warning message. Apparently, replaying the entire execution with the full log is an option but not the optimal one. For the event at 595007 to be correctly replayed, we need to replay events at 594804, 594825, ..., 594890, etc. Events from 4898 to 594803 are actually *irrelevant* to replaying the event at 595007. We constructed a new log by removing all the irrelevant events and then drove the replay with the reduced log. The same warning message was successfully reproduced. The execution was actually *fast forwarded* to the desired point by skipping the irrelevant part.

The EFF technique poses two challenges. The first one is how to identify and remove the irrelevant events; the second one is how to replay with the reduced event log. The following subsections describe how we handle these issues.

7.3 Event Dependence Graph

In dynamic slicing, given a value that is observed to be incorrect by the programmer (incorrect value may correspond to an incorrect output or a value that causes the program to crash). A set of executed statements that contributed to the value of the specified variable are computed as its dynamic slice. The executed statements not in the dynamic slice are not relevant to the investigated value. An analogous solution can be applied on the executed events to identify the set of irrelevant events for replaying a given execution region.

As an event usually corresponds to multiple executed statements, it is important to understand how we deal with events during the construction of a DDG. Since an event is usually handled inside the OS kernel, a tracing engine which runs in the application

space is not able to trace into the kernel. Hence the dependences within the event handler are not captured. The solution is to summarize the execution of an event into an abstraction, $E_j(U, D)$, according to the specifications of events. For instance, event "*n=read(fd, Buf, size)*" can be abstracted as "...(U = { *fd, seek_pointer(fd), size, Buf* }, D = { *seek_pointer(fd), Buf[0], Buf[1], ... Buf[n-1]* }. Note that only the first n elements of *Buf* are defined according to the specification of event *read*. This event both defines and uses the seek pointer of file *fd*.

An *event dependence graph* (EDG), can be constructed to reveal the dependences within events, which can be later on used to prune the irrelevant events.

Definition 7. *The **Event Dependence Graph** of a program run, EDG(N, E), consists of a set of nodes N and a set of directed edges E where: each node $n_i \in N$ corresponds to the i^{th} execution instance of event n in the program; and each edge $m_j \rightarrow n_i \in E$ denotes that there exists a dependence path from m_j to n_i, and there are no other executed events than m_j and n_i on the path.*

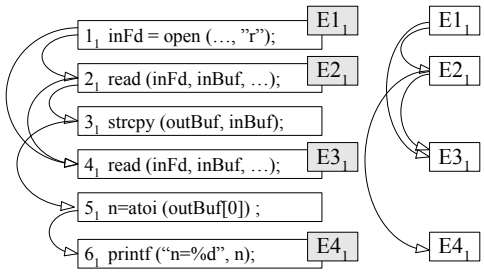


FIGURE 7.3. An example of dynamic dependence graph (DDG) and event dependence graph (EDG).

Figure 7.3 presents an example to illustrate a DDG and the corresponding EDG. The left hand side presents the DDG for the execution of a small piece of code. Statement executions 2_1 and 4_1 data depend on 1_1 because they use the file descriptor defined at 1_1. 4_1 data depends on 2_1 because 2_1 changes the file seek pointer. The graph on the right hand side shows the EDG. Event execution $E3_1$ depends on $E2_1$

because of the dependence path $2_1 \rightarrow 4_1$. Event execution $E4_1$ depends on $E2_1$ due to the dependence path $2_1 \rightarrow 3_1 \rightarrow 5_1 \rightarrow 6_1$. Note that the *read* events $E2$ and $E3$ are considered as different events because they occur at different program locations.

Control dependence between statements can also lead to dependence between events as demonstrated by another example in Figure 7.4, where event $E3_1$ depends on event $E2_1$ as the result of 30_1 control depending on 21_1 and 21_1 data depending on 20_1. The dependence between $E2_1$ and $E3_1$ belongs to control dependence as the execution of $E3_1$ is due to the result of $E2_1$. However, in EDGs data dependence and control dependence edges are not distinguished.

Precisely constructing the EDG requires accurately tracing each dependence (data, control, and potential). According to the previous experience, exactly tracing data and control dependences on the fly results in a slow down of up to two orders of magnitude. Thus, building precise EDG is a luxury that becomes worthy only when the cost can be amortized by a large number of replays. Otherwise, programmers would rather replay the entire log, which is equivalent to doubling the execution time, than endure the two orders of magnitude slow down in the first place and attain speed up in just a few replays later on. To address this issue, we have to be conservative by constructing an approximate EDG, in which one event depends on the other if and only if they are related by a *static* dependence path. In other words, only a static dependence graph is demanded, instead of a dynamic one, together with the event log to build the approximate EDG. The only runtime overhead paid is for event logging, which is significantly cheaper than tracing each dependence. Because the dependences between events are usually much simpler than the dependences between normal statements, which can be highly complicated due to pointer aliasing, being conservative in EDG construction introduces much less imprecision compared to being conservative in building DDG.

7.3.1 Meta Slicing on Event Log

Similar to dynamic slicing, given an EDG and an event, which the programmer wants to reproduce, meta slicing on the EDG computes the set of events that are needed in order to replay the given event.

Definition 8. *Given EDG(N, E), an event dependence graph, the* ***Meta Slice*** *of $e_i \in N$ denoted by MS(e_i) is the subgraph of EDG(N, E) which includes e_i as well as all other nodes and edges from which e_i is reachable, i.e.*

$$MS(e_i) = (\{e_i\}, \{e | e = m_j \rightarrow e_i \in E\}) \cup \bigcup_{\forall m_j \rightarrow e_i} MS(m_j)$$

For example in Figure 7.3, MS$(E4_1) = \{E1_1, E2_1, E4_1\}$. Note that we ignore the edges in MS for simplicity. We need to replay $E1_1$, which opens the file, and $E2_1$, which reads some data from the file, in order to correctly replay $E4_1$, which prints some value resulted from the computation over the input data. In Figure 7.4, MS$(E3_1) = \{E1_1, E2_1, E3_1\}$. $E2_1$ has to be replayed otherwise the control would not flow to $E3_1$.

```
...
10_1  inFd = open (path1, "r");  E1_1
...
20_1  n = read (inFd, buf, size);  E2_1
21_1  if (n!=size) {
        ...
30_1    inFd = open (path2, "r");  E3_1
        ...
40_1    S1;
        ...
      }
      ...
```

FIGURE 7.4. Another example of event dependence graph.

So far, how to find the set of relevant events in order to replay a given event has been discussed. However, in reality it could be a specific executed statement n_j that the programmer wants to replay. In this case, the set of closest events reachable from n_j in the DDG, denoted as ECut(n_j), need to be identified and then the meta slices

need to be computed on these events. For example in Figure 7.4, $\text{ECut}(40_1) = \{20_1\}$, the corresponding meta slice $\text{MS}(20_1) = \{10_1, 20_1\}$. Intuitively, both $E1_1$ and $E2_1$ need to be replayed in order to replay statement $S1$.

7.4 Replaying with A Reduced Event Log

It has been described how meta slicing can be applied to identify the set of events in the log that are relevant to replaying given part of the execution. However, meta slicing is not yet the ultimate solution. It is often the case that a meta slice can not be used directly to drive the replayed execution. For example, in Figure 7.3, $\text{MS}(E4_1) = \{E1_1, E2_1, E4_1\}$. Replaying with the meta slice fails because $E3_1$ was expected when the control flows to statement 4_1. This suggests that some events, even though irrelevant to replaying the desired part of the execution, cannot be pruned due to the control flow structure. Next, it will be explained how an event log is reduced with regard to the meta slice and the intrinsic control flow structure of the application.

```
5   gettimeofday( )
    …
10  while (1) {
      …
20    switch (c = getchar( )) {
        …
30      case 'a':
31        printf ("case A\n");
          … …
50      case 'c':
51        printf ("case C\n");
          … …
80      case 'o':
81        fd = open (..., "r");
          … …
90      case 'r':
91        n = read ( fd, buf, size);
92        if (n!=size) {
93          gettimeofday( )
94          printf ("Error: …\n");
          }
        }
    }
```

Event Log

```
5_1  gettimeofday
20_1 getchar
31_1 printf ("..A..")
20_2 getchar
80_1 open
20_3 getchar
31_2 printf ("..A..")
20_4 getchar
51_1 printf ("..C..")
20_5 getchar
91_1 read
20_6 getchar
51_2 printf ("..C..")
20_7 getchar
51_3 printf ("..C..")
20_8 getchar
91_2 read
93_1 gettimeofday
94_1 printf ("Err…")
```

FIGURE 7.5. An example on reducing the event log. The shaded events are those in $\text{MS}(94_1)$.

Before presenting the algorithm, let us first study an example that clearly explains how it is made possible to reduce a log without losing the validity. In Figure 7.5, the program displayed in the left column takes user commands from *stdin*. Different actions are taken based on different commands. For instance, messages are printed on the screen if 'a'/'c' is pressed; a file is opened if 'o' is pressed; the opened file is read if 'r' is read; if the data read does not match the size required, an error message is delivered. The event log for a particular execution is presented in the right column. During the execution, a file is opened and then read for twice; the second read does not satisfy the size wanted such that an error message is printed at 94_1; in between of these events, a number of events happen as the results of 'a'/'c' being pressed. Let us assume 94_1 is the event we want to replay. MS(94_1) is denoted as the shaded events in the log. Apparently, the meta slice is not legitimate for replay as event 5_1(*gettimeofday*), which is not in the meta slice, is expected at the beginning of the replayed execution. While 5_1 is not removable, events 20_1 and 31_1 can be removed without any problem. The important observation here is that 20_2 and 20_1 are *compatible* and thus 20_2 can be moved up to replace 20_1 such that the event in between, 31_1, is pruned.

Definition 9. *An event execution e_i is compatible with another event execution e_j iff their calling contexts are identical and they occur at the same program point.*

All the events 20_x in Figure 7.5 are compatible to each other. This example suggests we are able to alter the replayed execution by replacing an event with its compatible peer. The algorithm to reduce a log given the meta slice is presented as follows. Get_next_event() gets the next event from the log file; get_next_marked_event() gets the next event belongs to the meta slice, which we assume is precomputed, in the log file. These two methods share the same file seek pointer, which can be set by set_file_pointer(...).

```
Input:   the original log Log
Output:  the reduced log RLog
Initialize: RLog ← φ
while (e_m=get_next_marked_event(Log))!=EOF do
  e=get_next_event(Log)
  for each e_t from e to e_m in Log do
      if e_t.context ≡ e_m.context then
          goto L_1
      endif
      Rlog ← Rlog · e_t
  endfor
L_1:
  Rlog ← Rlog · e_m
  set_file_pointer(Log, e_m)
endwhile
```

The basic idea of the algorithm is that given a marked event e_m, an event in the meta slice, the earliest compatible event e_t is found in between e and e_m s.t. moving e_m up to replace e_t maximizes the savings. All the events between e and e_t including e are copied to the new log to satisfy the control flow structure confinement. The events between e_t and e_m are discarded.

Table 7.1 presents the reduction procedure of the example in Figure 7.5. As shown in the table, during iteration one, 5_1 is the first event retrieved from the log, and 20_2 is the first marked event. 20_2 can be moved up to replace 20_1 such that 5_1 and 20_2 are the two events appended to the new log. During the second iteration, 80_1 is the next event and also the next marked event such that it is simply copied to the new log. In iteration three, moving 20_5 up to replace 20_3 results in cutting the events from 20_3 to 50_1. The final reduced log is shown in the last row of the table. The reduce log can be used to drive the replayed execution to reproduce the error message at 94_1.

7.5 Experimental Results

A few issues need to be address in order to carry out the evaluation. The first issue is that what benchmarks should be used. The selected programs should be able to

TABLE 7.1. Computation table for figure 7.5.

Iteration	e	e_m	RLog
1	5_1	20_2	$5_1\ 20_2$
2	80_1	80_1	$5_1\ 20_2\ 80_1$
3	20_3	20_5	$5_1\ 20_2\ 80_1\ 20_5$
4	91_1	91_1	$5_1\ 20_2\ 80_1\ 20_5\ 91_1$
5	20_6	20_8	$5_1\ 20_2\ 80_1\ 20_5\ 91_1\ 20_8$
6	91_2	91_2	$5_1\ 20_2\ 80_1\ 20_5\ 91_1\ 20_8\ 91_2$
7	93_1	94_1	$5_1\ 20_2\ 80_1\ 20_5\ 91_1\ 20_8\ 91_2\ 93_1\ 94_1$

run for a long time. We looked at the set of bugs studied in [59, 67, 65] and picked the programs that can execute for a long time. Table 7.2 presents the set of selected programs. Most of them are user interactive programs. The second issue is how to obtain the input that can drive the execution for a long time and then crash the execution. On the other hand, the execution should not be so long that it becomes too heavy a task to collect the data. Unfortunately, the input coming with the selected bugs usually leads to very short executions. Given the fact that most benchmarks are interactive, a long input was constructed by first performing a lot of user actions and then apply the failure inducing input –the input comes with the benchmarks. For example in *mutt*, the following actions were taken: (i) opening an email account; (ii) going through all the emails one by one, the total is about six hundreds; (iii) trying to switch to an invalid folder; repeating steps (ii) and (iii) two more times; providing the failure inducing input and crashing the program. The user time was collected as the performance indicator since the real time may significantly differ each time depending on the user's behavior.

Four execution scenarios were investigated: *orig.* denotes the original execution; *traced* denotes the original execution plus the dependence tracing; *logged* represents the original execution plus logging; *EFF* represents the fast forwarded execution plus the dependence tracing. In the logged run, an event log is created. The EFF technique

TABLE 7.2. Description of the benchmarks

Benchmark	Description	LOC	Bug Type
bc-1.06	interactive calculator	14.4K	heap overflow
mc-4.5.55	file manager	86.2K	stack overflow
mutt-1.4.2.1i	email client	453.6K	heap overflow
pine-4.44	email client	211.9K	stack overflow
pine-4.44	email client	211.9K	heap overflow
squid-2.3	web proxy cache server	93.5K	heap overflow

is applied to reduce the log. The statement instance we want to replay is where the crash happened. The EFF technique is able to reproduce the crash in a much shorter execution. Due to the complexity of the system, the implementation is not sound at the current stage. Some times a few event dependences had to be hard coded, otherwise the reduced log was not valid to drive the replay which was manifested as an event missing when it was expected or the presence of an extra event.

TABLE 7.3. Performance comparison of different execution scenarios.

Benchmark	Orig. (sec.)	Traced (sec.)	Traced /Orig.	Logged (sec.)	Logged /Orig.	EFF (sec.)	Traced /EFF
bc-1.06	13.6	2040.4	150.6	16.2	1.19	0.05	40808.8
mc-4.5.55	10.28	417.8	40.64	13.47	1.31	0.05	8356
mutt-1.4.2.1i	19.7	3237.7	164.5	26.1	1.32	0.06	53960.8
pine-4.44(stack)	14.4	2088.4	145.1	36.8	2.55	0.12	17403.6
pine-4.44(heap)	13.9	2102.2	151.5	34.4	2.47	0.20	10510.9
squid-2.3	14.6	1131.6	77.3	25.6	1.75	0.17	6656.4

Table 7.3 compares the performance under the four scenarios. The original runs, which were terminated by crashes, consume execution times ranging from 10.2 to 19.7 seconds, which correspond to the real times of a few minutes. They are not long by simply looking at the raw numbers, but they well exceed the capability of the dependence tracing technique. The executions can be easily extended by repeating the user actions. The side effect is the increased difficulty of collecting the time for the executions in the *traced* scenario. Note that even though checkpointing is supported in our system, the original execution does not last long enough to trigger

it. Fortunately, it does not affect the evaluations of the EFF technique and the effectiveness of dynamic slicing on long running programs. From table 7.3, we have the following observations.

- Dependence tracing introduces 40.46 to 164.5 times slow down. A programmer may accept it for a short run but highly unlikely for a long run.
- The slow down factors for logging range from 1.19 to 2.55, which are significantly smaller than the tracing slow down factors. For user interactive programs, the overhead is not noticeable.
- EFF can greatly shorten the executions such that dependence tracing becomes bearable.

TABLE 7.4. Comparison of the event logs.

Benchmark	# of events in Orig.	# of events in EFF	Orig./EFF
bc-1.06	340509	7	48644.0
mc-4.5.55	322172	16020	20.1
mutt-1.4.2.1i	262559	489	536.9
pine-4.44	7365830	3028	2432.6
pine-4.44	8707316	27279	319.2
squid-2.3	1620988	795	2038.9

TABLE 7.5. Comparison of the dependence graphs.

Benchmark	# of dep. in Orig.	# of dep. in EFF	Orig./EFF
bc-1.06	2.18×10^{10}	4.9×10^{5}	44489.8
mc-4.5.55	0.69×10^{10}	9.6×10^{7}	71.8
mutt-1.4.2.1i	4.86×10^{10}	4.21×10^{7}	1154.4
pine-4.44	1.95×10^{10}	2.68×10^{7}	727.6
pine-4.44	2.78×10^{10}	1.55×10^{8}	179.4
squid-2.3	1.1×10^{10}	1.93×10^{6}	5699.5

Table 7.4 compares the numbers of events before and after event reduction. The reduction factors range from 20.1 to 48644.0, which very well explain why the fast

forwarded executions become so short. Table 7.5 presents the numbers of the exercised data dependences in the original and the fast forwarded executions. Note that these numbers are collected after the intra-basic-block optimization as introduced in chapter 3 which eliminates considerable redundant dependences. The numbers for the fast forwarded executions are much smaller. The constructed dependence graphs can be stored even without further compression as discussed in chapter 5.

7.6 Summary

Dynamic slicing can be enabled on a set of long running programs by developing a novel execution fast forwarding technique. Fast forwarding can be achieved by driving the replay with a reduced event log file. Given a desired execution region, a large portion of the events are not relevant to replaying it. Meta slicing is designed to eliminate this redundancy in the log file. With the execution fast forwarding technique, the replayed execution becomes substantially shorter and yet the wanted execution region is precisely reproduced. The reduction factors of the sizes of dynamic dependence graphs range from 179.4 to 44489.8. As a result, dynamic slicing can be practically applied to isolate the cause effect chain leading to the failure.

Chapter 8
Related Work

8.1 Profiling

Program profiles for realistic program runs can greatly benefit applications such as compiler optimization, architecture simulation and fault locations. This is because program profiles can be analyzed to identify program characteristics that can then be exploited by researchers to guide designs of superior compilers and architectures or be analyzed by programmers to nail program bugs. The key challenge is that the amounts of profile information generated during realistic program runs can be extremely large.

One approach to reducing the amount of profile data is by using lossy compression or summarization techniques. Lossy compression of variety of profiles has been carried out including, dynamic dependence profiles in [7], dynamic control flow in [9], and dynamic values in [19]. Although for many applications summarization is adequate, for others they have proved to be inadequate. For example, it has been shown that summarization of dynamic data dependences results in high levels of inaccuracy in dynamic data slices [87].

Researchers have developed lossless compression techniques to limit the space required to store different types of profiles. Lossless compression techniques for several different types of profiles have been separately studied. Compressed representations of *control flow* traces can be found in [53, 89]. These profiles can be analyzed for presence of hot program paths or traces [53] which have been exploited for performing path sensitive optimizations [79, 15, 32] and path-sensitive prediction techniques [45]. *Value profiles* have been compressed using value predictors [16] and used to perform code specialization [19], data compression [90], value speculation [56], and value en-

coding [78]. *Address profiles* have also been compressed [21] and used for identifying hot data streams that exhibit data locality which can help in finding cache conscious data layouts [68] and developing data prefetching mechanisms [22, 47].

Compared to the individual types of profiles, the unified WET representation enhanced with the tier compression strategy described in chapters 3 and 5 provides high compression rate and easy access to multiple types of profiles. This greatly benefits dynamic slicing based fault location techniques. Moreover, it leads to exploration of advanced compiler and architecture techniques which simultaneously exploit multiple types of profiles.

Note that in [13], Bhansali et al. use a fine grained checkpointing mechanism which is able to reproduce different kinds of execution profiles. They do not store the complete profiles, instead, they store the minimal amount of profile information, which is a subset of the complete load value trace, in order to replay the execution and retrieve other profiles. They achieve the space efficiency of 0.1 bit per instruction. This approach essentially provides the capability of random re-execution at certain level, which does not directly serve the demands of many applications. For example, in the application of dynamic slicing, even though re-execution is able to recover dependences inside the re-executed window, how to process and store these dependences is still an issue that needs to be addressed. In other words, their technique and our WET representation can be complementary.

8.2 Fault Location

The other main focus of this dissertation is on fault location. The work related to this dissertation is presented in the following subsections.

8.2.1 Slicing Based Approaches

Dynamic slicing was introduced as an aid to debugging by Korel and Laski in 1988 [49]. Ever since then, dynamic slicing has been studied by many researchers [6, 48, 50, 42, 64, 73]. Agrawal et al. [42] proposed subtracting a single correct execution trace from a single failed execution trace. In [64], Pan and Spafford presented a family of heuristics for fault localization using dynamic slicing. Compared to these previous works, this dissertation is the first one to compare the effectiveness of dynamic slicing algorithms in fault location.

Since the main idea of slicing is to focus the user's attention on a relevant subset of statements in the program, it is only natural that researchers have explored techniques for narrowing the relevant set of statements beyond what is contained in a single slice. In this dissertation, novel slicing criteria are identified whose dynamic slices are highly effective in capturing faulty code and therefore their intersection also captures the faulty code. In [34], a *dynamic chop*, which is the dynamic dependence subgraph between two nodes, is used to derive dynamic path conditions. A constraint solver is then used to test whether the derived conditions can be satisfied. If so, the resolved input serves as a witness to the failure. If not, there is no dependence between the two nodes even though there exists a dependence path between them. In [20] difference of backward slices is computed with the aim of eliminating those statements that are less likely to be faulty from the backward slice of an erroneous output. While [20] is a set based technique, the confidence analysis presented in chapter 6 is a fine grained graph based pruning technique, which provides the capability of discretely pruning multiple instances of a static statement. In addition, confidence analysis, for the first time, considers the mappings between executed statement instances when computing the likelihood of a statement instance being faulty.

8.2.2 Statistical Approaches

Recently a large body of research has been focused on the use of statistical techniques for fault location [66, 46, 57, 55, 57, 27]. Harrold et al. [36] compared the spectra of passing and failing runs and found that failing runs tend to have unusual coverage spectra. Jones et al. [46] ranked each statement according to its ratio of failing tests to correct tests and used this information to assist fault location. Liblit et al. [55] describe a sampling framework and present an approach to guess and eliminate predicates to isolate a deterministic bug. For isolating nondeterministic bugs, they use statistical regression techniques to identify predicates that are highly correlated with the program failure. Liu et al. [57] present a more accurate statistical model which eliminates some of the limitations in [55] by considering the situation of only some executed *instances* of a single predicate being faulty. Fei and Midkiff [27] present an online bug detecting technique, in which a correct model is built by training through a set of correct runs, and any significant deviation from this model in the detection run raises a flag. Renieris and Reiss [66] focus on the difference between the failing run and a *single* passing run with similar spectra as a means to narrow down the search space for faulty code. Xie and Engler [77] show that many redundancies in programs correspond to hard program errors. Hangal and Lam [35] identified the causes of some programming errors in Java programs by observing violations of program invariants.

Dynamic slicing differs from statistical debugging techniques in several significant ways and to some extent it is complimentary to statistical techniques. Statistical debugging techniques rely on dynamic information (e.g., patterns of predicate outcomes) collected for a certain number of program runs. In contrast, in dynamic slicing, all slices are based upon the dynamic dependence graph of a single failed program run. Statistical techniques have the capability of predicting the future happening of a failure while dynamic slicing is essentially a post-mortem analysis of the failure. On the other hand, the ability to predict in statistical techniques comes with the cost of false

positives, i.e., some correct statements are indicated as faulty. Another important characteristic of statistical techniques is that they usually rank program statements according to a score which captures the likelihood of the statement representing faulty code. The programmer can then examine the statements in the order of ranking to locate faulty code. Ranking can also be used in conjunction with dynamic slicing. In [52, 88] the statements in backward dynamic slices are ranked according to their dependence distances from the point at which erroneous output is observed and in confidence analysis they are ranked according to *confidence* values which measure the likelihood that they produced correct results. Finally, statistical techniques usually simply provide a ranked fault candidate set. In the procedure of debugging, the programmer usually follows certain cause effect relations to locate the bug instead of inspecting individual statements one by one. In contrast, dynamic slicing usually produces a dynamic dependence graph, the dependence chains which essentially express the cause-effect relations.

8.2.3 State Based Approaches

Zeller has presented a series of techniques [41, 81, 24] from isolating the failure inducing input to isolating cost-effect chains in both space and time. The basic idea is to find the specific part of the *input/program state* which is critical to the program failure by minimizing the difference between the *input/program state* leading to a passing run and that leading to a failing run. Techniques presented in this dissertation can be combined with Zeller's technique in many aspects, for instance, the isolated *causes* are perfect slicing criteria starting from which dynamic slicing may produce a much smaller fault candidate set than from the failure point.

In [39], He and Gupta present a technique which systematically searches for the fix to a program error. Given the trace of a failed execution and the post-condition, the technique traverses the trace backward and identifies the execution points at which

the actual program state deviates from the state inferred from the post-condition. The technique then automatically searches for possible modifications to the program which can make the two program states consistent. These modifications are further validated using other test cases.

Some of the techniques presented in this dissertation, such as predicate switching, also approach the problem of fault location through program state investigation and manipulation.

8.2.4 Static Analysis Based Approaches

The techniques presented in this dissertation are dynamic techniques. There is a large group of static techniques for fault location. The philosophy of static fault location is to first construct a correct model by specifying certain rules, and then statically analyze the target program to see if these rules are strictly followed. *LCLint* [26] uses annotations to represent assumptions about function interface, variables, and type explicit. Constraints derived from these annotations are checked at compile time. Any violations are considered as potential errors. *CQaul* [29] proposes a technique in which users annotate their programs with flow sensitive type qualifiers. The correctness of the programs can be checked by inference. *Prefix* [18] presents an important methodology of using static program analysis to detect memory related program errors. The idea is to symbolically execute the functions, modeling the memory and reporting any inconsistencies. *Slam* [10] uses techniques from program analysis, model checking, and automated deduction to check whether a C program follows certain rules in using an API. *Blast* [40] is a similar technique which performs model checking on the safety properties of C programs. In [44], the code of a procedure is modeled as relational formulas, which are conjoined with the negation of the procedure's specification. A constraint solver is used to either verify the procedure or generate counter-examples. In [76], Xie and Aiken translates C programs into boolean formulas such that boolean

satisfiability solvers can be used to efficiently check any violations to certain specified properties.

In [60], Manevich et al. propose using post-mortem static analysis to locate faults. The idea is that after knowing the type of failure and the program location where the failure happened, static analysis such as pointer analysis can be performed more effectively such that the flow of a value can be more accurately pin-pointed. This technique reveals an interesting direction of combining both dynamic and static analysis in fault location.

Compared to dynamic approaches, static techniques usually require programmers to write annotations or specifications. A large number of false positives are usually incurred by the conservative nature of underlying static analysis such as pointer analysis.

To address some of these existing issues, a new stream of research has been conducted on using data mining techniques to automatically discover specifications. In [25], Engler et al. propose inferring rules, called program beliefs, from a large pool of source code. These beliefs are crosschecked and contradictions are reported. While [25] only captures pair-wise programming rules, Li and Zhou [54] improve this technique by mining more complex rules. In [8], Ammons et al. propose discovering formal specifications from executions instead of source code. *Dynamine* [58] identifies highly correlated method calls as well as common bug fixes by mining source code check-ins.

Chapter 9
Conclusions

9.1 Contributions

This dissertation makes contributions in the area of dynamic slicing. In particular, it greatly improves the efficiency and effectiveness of dynamic slicing. As a debugging aid, dynamic slicing techniques have been invented for a long time. However, most of the previous research was performed on toy programs and small runs, which diminished the enthusiasm about dynamic slicing. This dissertation discusses techniques that make dynamic slicing practical and effective in locating faults in realistic programs. More specifically, it provides answers to the following four research questions.

Q1: How expensive is precise dynamic slicing for real programs and realistic runs? This dissertation shows that existing dynamic slicing algorithms are quite expensive because they require constructing dynamic dependence graphs, which can take up to 2 gigabytes space, if fully constructed, for the execution of 130 millions intermediate statements for realistic programs. A demand driven strategy alleviates the space problem but the slicing times are very slow. Because in a demand driven algorithm, a partial dependence graph is constructed on demand in response to a slicing request. Computing multiple slices require repeatedly traversing the execution traces, which is a procedure that could take up to 20 minutes for a 130 millions intermediate statements run. To conclude, without sophisticated designs, existing dynamic slicing algorithms are very expensive in terms of space and time.

Q2: Can dynamic slicing be made practical? Precise dynamic slicing is so expensive because it constructs huge dynamic dependence graphs(DDGs). Traversing

through dynamic information contained in DDGs takes significant amount of time. Therefore, optimizations on dynamic dependence graphs, which reduce the sizes of generated graphs, save both space and time. The redundancies in DDGs can be eliminated by two means: a large portion of the dynamic dependence edges in a DDG can be replaced by static edges because they can be inferred from static edges; a DDG can be transformed to enable more inferences. The results show that after applying all the optimizations, only 6% of the original information need to be represented explicitly in a DDG. For the same 130 millions intermediate statements run, the optimized DDG takes the space of 94 megabytes on an average. Note that optimizations are different from compression. The latter incurs overhead in traversal while the former speeds it up. As shown by the experiments, it takes 16 seconds to traverse the optimized DDGs on average.

In order to further reduce the space consumption, compression techniques can be added on top of optimizations. A novel generic bidirectional stream compression technique is introduced, which can achieve high compression rate and at the same time is capable of traversing the compressed stream in both the forward and backward directions. The optimizations and compression are so effective that more information such as value profiles can be embedded in DDGs. The information can be used to compute more prolific slices such as slices annotated with the computed values and prune dynamic slices. A new dynamic representation – Whole Execution Traces(WETs), is proposed to represent control flow, dependence, and value profiles in a unified form. WETs can store the complete dynamic information of a 3.9 billion intermediate statements run into 2 gigabytes space. Due to the overhead introduced by compression, computation of a slice on WETs has a 6X slowdown, which is still much faster than a demand driven algorithm.

In order to scale the technique to long running programs, this dissertation also discusses using checkpointing in combination with dynamic slicing. Checkpoints are usually created in an interval of minutes while dynamic slicing, even after applying all

the proposed optimization and compression techniques, can only handle executions of up to a few seconds. An execution fast forwarding technique is proposed to significantly reduce the replay execution of a checkpoint interval such that dynamic slicing becomes applicable. To conclude, dynamic slicing becomes much more practical with all these techniques.

Q3: Is dynamic slicing really useful in debugging real software errors? Experimental results in this dissertation indicate that existing dynamic slicing algorithms are quite successful in containing the root causes of real software errors once they can be applied. However, they are not always applicable since a faulty program may not produce any output as a result of faulty code getting executed. In such cases, a wrong output value cannot be recognized, as a result, a slice cannot be computed. Moreover, existing algorithms usually produce large slices which require significant amount of time to manually inspect. Finally, there are cases in which conventional backward dynamic slicing techniques fail to capture the root causes due to the existence of potential dependences. Even though relevant slicing was proposed as a plausible solution, the conservative nature of this technique limits its success.

Q4: Can the fault location effectiveness of dynamic slicing be improved? The limitations originate from one fact – existing algorithms compute the backward slice(BwS) on only one type of evidence in a failed run, which is the wrong output. However, a software error may manifest itself in many different ways during a failed run. In other words, more evidences can be collected to help analyze the error. In this dissertation, new types of dynamic slices are proposed which take advantage of different types of evidences. A *forward slice* (FwS) is computed for *failure inducing input difference*, which is identified by the *delta debugging* technique. A *bidirectional slice* (BiS) is computed for a *critical predicate*, switching the outcome of which produces the expected correct output. The experimental results show that while each

of BwS, FwS, and BiS may not be applicable for all the real errors under consideration, each error can be captured by at least one type of slice. The three types of slices are all applicable for most of the errors, which gives rise to the opportunity of combining them together. Considering the wrong output, minimal failing inducing input difference, and critical predicates together reduce the fault candidate set to 36% of the smallest of the three sets computed by considering individual evidences. For some errors, even after combining multiple types of slices may still include a lot of statements which are highly unlikely to be wrong. These statements can be pruned from the slice by considering the values produced by their executions. A new dynamic analysis, *confidence analysis*, is proposed to take advantage of value profiles. The analysis computes an estimate for the likelihood of a statement execution being faulty. The statements that are considered correct by the analysis are pruned from the slice. This analysis can reduce a backward dynamic slice to 41% on average.

9.2 Future Directions

Improving relevant slicing through predicate switching. Execution omission errors are known to be resistant to conventional backward dynamic slicing. These errors lead to failure at runtime by means of certain statements not being executed while they should have been if there were no errors, which contradicts the fact that dynamic slicing techniques are mostly based on the information collected from executed statements. Although researchers have attempted to tackle this problem through relevant slicing, the static nature of this technique becomes a barrier to success. Based on the observation that these errors are hard to detect because some of the dependences are invisible, it is possible to enforce the execution of the omitted code by switching predicates such that those implicit dependences become tractable. Once these hidden dependences are disclosed, dynamic slices can be computed and effectively pruned to produce fault candidate sets containing the execution omission errors.

Dynamic slicing multithreaded programs. As multi-core and shared memory systems are becoming more and more popular, multithreading becomes a very important programming model. Compared to sequential programs, multithreaded programs are a lot harder to debug because an error may be caused by the interactions between threads. This also implies that a good automatic debugging technique is much more demanded for multithreaded programs. This dissertation shows that dynamic slicing is very effective in locating faults in sequential programs. It would be very interesting to see how dynamic slicing can be applied on multithreaded programs.

Dynamic matching based on WET. In many application areas, including the areas of software debugging, maintenance, and piracy detection, situations arise in which there is a need for comparing two versions of a program. An existing class of algorithms that compare two program versions are static differencing algorithms. While these algorithms report static differences between code sequences, in situations where the two program versions correspond to the original and transformed versions of a program, it is desirable to match code sequences that dynamically behave the same even though they statically appear to be different. Let us consider the applications such as software piracy detection and debugging of optimized code. In these two applications, one program version is created by transforming the other version. In the first application, code obfuscation transformations may have been performed to hide piracy. In the second application, transformations are applied to generate an optimized version from the unoptimized version. Since the transformed (obfuscated or optimized) code looks very different from the original code, static differencing approaches will not work for the above applications. The WET representation proposed in this dissertation efficiently captures the complete dynamic information of an execution and thus could serve as a basis for dynamic matching techniques.

Other applications of WET. Collection, maintenance, and analysis of detailed program profiles for realistic program runs can greatly benefit compiler and architecture researchers. This is because program profiles can be analyzed to identify program characteristics that can then be exploited by researchers to guide the design of superior compilers and architectures. While individual types of profiles have been studied, the correlations of multiple types of profiles and their effects on architecture and compiler design remain unexplored. For example, hardware predictors usually predict based on control flow histories. It would be interesting to study whether considering value profiles in the mean time can improve the prediction accuracy.

References

[1] http://valgrind.org.

[2] http://www.cse.unl.edu/∼galileo/sir.

[3] http://www.elis.ugent.be/diablo/.

[4] http://www.trimaran.org.

[5] Information week. *Issue on Software Quality*, 2002.

[6] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, 1993.

[7] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, New York, United States, 1990.

[8] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, Portland, Oregon, 2002.

[9] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, Paris, France, 1996.

[10] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, Toronto, Ontario, Canada, 2001.

[11] Arpad Beszedes, Csaba Farago, Zsolt Mihaly Szabo, Janos Csirik, and Tibor Gyimothy. Union slices for program maintenance. In *ICSM '02: Proceedings of the IEEE International Conference on Software Maintenance*, Montreal, Canada, 2002.

[12] Arpad Beszedes, Tamas Gergely, Zsolt Mihaly Szabo, Janos Csirik, and Tibor Gyimothy. Dynamic slicing method for maintenance of large c programs. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 105–113, Lisbon, Portugal, 2001.

[13] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinic, Darek Mihocka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Virtual Execution Environments Conference*, pages 154–163, Ottawa, Canada, 2006.

[14] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 357–363, Santa Barbara, California, 1995.

[15] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 1–14, Montreal, Quebec, Canada, 1998.

[16] Martin Burtscher and Metha Jeeradit. Compressing extended program traces using value predictors. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 159–169, New Orleans, Louisiana, 2003.

[17] Martin Burtscher and Benjamin G. Zorn. Exploring last n value prediction. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 66–76, Newport Beach, California, 1999.

[18] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[19] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–269, Research Triangle Park, North Carolina, United States, 1997.

[20] T. Y. Chen and Y. Y Cheung. Dynamic program dicing. In *ICSM '93: Proceedings of the IEEE International Conference on Software Maintenance*, pages 378–385, Montreal, Quebec, Canada, 1993.

[21] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 191–202, Snowbird, Utah, United States, 2001.

[22] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN*

2002 Conference on Programming Language Design and Implementation, pages 199–209, Berlin, Germany, 2002.

[23] Andy Chow Chou. *Static analysis for bug finding in systems software*. PhD thesis, Standford University, 2003.

[24] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the International Conference on Software Engineering*, pages 342–351, St. Louis, MO, USA, 2005.

[25] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP'01: Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pages 57–72, Chateau Lake Louise, Banff, Canada, 2001.

[26] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, United States, 1996.

[27] Long Fei and Samuel P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 84–95, Ottawa, Canada, 2006.

[28] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[29] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, 2002.

[30] Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. Differential fcm:increasing value prediction accuracy by improving table usage efficiency. In *HPCA'01: Procedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 207–216, Monterrey, Mexico, 2001.

[31] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, Long Beach, CA, USA, 2005.

[32] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial redundancy elimination using speculation. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–115, San Francisco, California, 1997.

[33] Tibor Gyimothy, Arpad Beszedes, and Istan Forgacs. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303–321, Toulouse, France, 1999.

[34] Christian Hammer, Martin Grimme, and Jens Krinke. Dynamic path conditions in dependence graphs. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial evaluation and Semantics-based Program Manipulation*, pages 58–67, Charleston, South Carolina, 2006.

[35] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the International Conference on Software Engineering*, pages 291–301, Orlando, Florida, 2002.

[36] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[37] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, Montreal, Quebec, Canada, 1998.

[38] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, Boston, MA, USA, 2004.

[39] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *FASE'04: Proceedings of Fundamental Approaches to Software Engineering*, pages 267–280, Barcelona, Spain, 2004.

[40] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with blast. In *SPIN'03:Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, pages 235–239, Portland, Oregon, 2003.

[41] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 135–145, Portland, Oregon, United States, 2000.

[42] Saul London W. Eric Wong Hiralal Agrawal, Joseph R. Horgan. Fault localization using execution slices and dataflow tests. In *ISSRE'95: Proceedings of the Sixth IEEE International Symposium on Software Reliability Engineering*, pages 143–151, Toulouse, France, 1995.

[43] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, 1994.

[44] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 14–25, Portland, Oregon, United States, 2000.

[45] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-based next trace prediction. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 14–23, Research Triangle Park, North Carolina, United States, 1997.

[46] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the International Conference on Software Engineering*, pages 467–477, Orlando, Florida, 2002.

[47] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the International Symposium on Computer Architecture*, pages 252–263, Denver, Colorado, United States, 1997.

[48] M. Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linkoping University, 1993.

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

[50] Bogdan Korel and Juergen Rilling. Application of dynamic slicing in program debugging. In *AADEBUG'97: Proceedings of the International Symposium on Automated Analysis-driven Debugging*, pages 43–58, Linkping, Sweden, 1997.

[51] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *ISSTA '94: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 66–79, Seattle, Washington, United States, 1994.

[52] Jens Krinke. Visualization of program dependence and slices. In *ICSM '04: Proceedings of the IEEE International Conference on Software Maintenance*, pages 168–177, Chicago, USA, 2004.

[53] James R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming language Design and Implementation*, pages 259–269, Atlanta, Georgia, United States, 1999.

[54] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, Lisbon, Portugal, 2005.

[55] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, San Diego, California, USA, 2003.

[56] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, Paris, France, 1996.

[57] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295, Lisbon, Portugal, 2005.

[58] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, Lisbon, Portugal, 2005.

[59] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: a benchmark for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, Illinois, 2005.

[60] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: explaining program failures via postmortem static analysis. In *FSE-12: Proceedings of the Twelfth ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 63–72, Newport Beach, CA, USA, 2004.

[61] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, Madison, Wisconsin, USA, 2005.

[62] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In *DCC '97: Proceedings of the Conference on Data Compression*, pages 3–11, Washington, DC, USA, 1997. IEEE Computer Society.

[63] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *ICSE '99: Proceedings of the International Conference on Software Engineering*, pages 422–431, Los Angeles, California, United States, 1999.

[64] Hsin. Pan and Eugene H. Spafford. Heuristics for automatic localization of software faults, 1992. Technical Report SERC-TR-116-P, Purdue University.

[65] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP'05: Proceedings of the Twentith ACM Symposium on Operating Systems Principles*, pages 235–248, Brighton, UK, 2005.

[66] Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *ASE '03: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, 2003.

[67] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI '04: Proceedings of the the Sixth Symposium on Operating System Design and Implementation*, pages 303–316, San Francisco, California,, 2004.

[68] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 140–153, Portland, Oregon, 2002.

[69] Joseph R. Ruthruff, Margaret Burnett, and Gregg Rothermel. An empirical study of fault localization for end-user programmers. In *ICSE '05: Proceedings of the International Conference on Software Engineering*, pages 352–361, St. Louis, MO, USA, 2005.

[70] Yiannakis Sazeides. Instruction-isomorphism in program execution. In *Proceedings of the 1st Annual Value Prediction Workshop*, San Diego, CA, 2003.

[71] Yiannakis Sazeides and James E. Smith. Implementations of context-based value predictors. Technical Report TRECE -97-8, University of Winsconsin.

[72] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258, Research Triangle Park, North Carolina, United States, 1997.

[73] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE'04:Proceedings of the International Conference on Software Engineering*, pages 512–521, Edinburgh, United Kingdom, 2004.

[74] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979. University of Michigan.

[75] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the International Conference on Software Engineering*, pages 439–449, San Diego, California, United States, 1981.

[76] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, Long Beach, California, USA, 2005.

[77] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60, Charleston, South Carolina, USA, 2002.

[78] Jun Yang and Rajiv Gupta. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems*, 1(1):79–105, 2002.

[79] Cliff Young and Michael D. Smith. Better global scheduling using path profiles. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE International Symposium on Microarchitecture*, pages 115–123, Dallas, Texas, United States, 1998.

[80] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, 1999.

[81] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, Charleston, South Carolina, USA, 2002.

[82] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

[83] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[84] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceeding of the International Conference on Software Engineering*, pages 272–281, Shanghai, China, 2006.

[85] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 197–206, Lisbon, Portugal, 2005.

[86] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the International Conference on Software Engineering*, pages 319–329, Portland, Oregon, 2003.

[87] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27(4):631–661, 2005.

[88] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the International Symposium on Automated Analysis-driven Debugging*, pages 33–42, Monterey, California, USA, 2005.

[89] Youtao Zhang and Rajiv Gupta. Timestamped whole program path representation and its applications. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 180–190, Snowbird, Utah, United States, 2001.

[90] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 14–28, London, UK, 2002.

[91] Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel. Midkiff, and Josep Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO 37: Proceedings of the 37th annual ACM/IEEE International Symposium on Microarchitecture*, pages 269–280, Portland, OR, 2004.

[92] Craig B. Zilles and Gurindar S. Sohi. Understanding the backward slices of performance degrading instructions. In *ISCA '00: Proceedings of the International Symposium on Computer Architecture*, pages 172–181, Vancouver, British Columbia, Canada, 2000.