Fault Location and Avoidance in Long-Running MultiThreaded Applications

by

Sriraman Tallam

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 7

Statement by Author

This thesis is dedicated to my parents, *Shantha Kannan* and *Kannan Narasimhachari*.

Acknowledgements

I am deeply indebted to my advisor, *Prof. Rajiv Gupta*, for the last 5 years of my academic career. His work ethic and humility make him a role model and I am very fortunate and proud to have worked with him. He motivated me towards research and taught me how to do quality research. It was through a lot of his efforts that we could together publish in reputed conferences and journals. I sincerely thank him for all he has done to make the years in Arizona the best of my entire student life.

I would then like to thank *Dr. Neelam Gupta* for her efforts in helping me do good research. She made me realize the rigor involved in performing empirical research and did bear with me during my most difficult times. The research I have done with her has given me a lot of satisfaction.

I also want to thank *Dr. Xiangyu Zhang* who was my group mate in Arizona and who is now a faculty member in Purdue. I will always remember him as my other advisor. He is a great role model for every aspiring graduate student and I was never short of motivation when he was around. He has helped me a lot throughout my PhD and has had a great influence on me.

I would like to thank *Dr. John Kececioglu* with whom I have spent a lot of time learning algorithms. I will remember his courses as the most challenging and satisfying ever. I also would like to thank him for serving on my dissertation commitee and providing useful feed-back. I would also like to thank *Dr. Kobus Barnard* for his feed-back as my dissertation committee member.

I would like to thank *Chen Tian*, also my group mate in Arizona, for working with me on a few papers and producing great work. I would also like to thank my other group mates : *Arvind Krishnaswamy, Bengu Li, Vijayanand Nagarajan, and Dennis Jeffrey* for their help. I also want to thank my friends *Hariharan Lalgudi, Praveen Rao, Krishna Muralidharan, Ranjini Swaminathan, Eagu Kim, Rui Zhang,* *Somasundaram Perianayagam, Mohan Rajagopalan, Haifeng He.* I had such a good time with them in Arizona. I want to thank my friends *Michael Isaacs* and *Raji Baskaran* who have helped me a lot right from my student days in India.

I want to thank my parents for so much that they have provided me with and my brothers for their support.

Last but not the least is thanks to my wife *Aarthi.* We got married when I was still a student in Arizona and she made the last year of my PhD so memorable.

Table of Contents

Table of Contents—*Continued*

Table of Contents—*Continued*

List of Figures

List of Figures—*Continued*

List of Tables

Abstract

Faults are common-place and inevitable in complex applications. Hence, automated techniques are necessary to analyze failed executions and debug the application to locate the fault. For locating faults in programs, dynamic slices have been shown to be very effective in reducing the effort of debugging. The user needs to inspect only a small subset of program statements to get to the root cause of the fault. The dynamic slice connects the various executed instructions through dependences and the root cause of the fault is found by inspecting the instructions in the dynamic slice starting from the point where the execution failed. While prior work has primarily focussed on single-threaded programs, this dissertation shows how dynamic slicing can be used for fault location in multithreaded programs. This dissertation also shows that dynamic slices can be used to track down faults due to data races in multithreaded programs by incorporating additional data dependences that arise in the presence of many threads. It shows how the dynamic slices of multithreaded programs are represented and how they are traversed for fault location. Case studies presented show that, using dynamic slices, less than 5 program statements had to inspected to discover the root cause of the bug.

In order to construct the dynamic slices, dependence traces (control and data) are collected and processed. However, program runs generate traces in the order of Gigabytes in a few seconds. Hence, for multithreaded program runs that are long-running, the process of collecting and storing these traces poses a significant challenge. This dissertation proposes two techniques to overcome this challenge. First, a new trace representation is proposed to store the generated control and data dependence traces compactly on disk. Second, schemes that can reduce the size of the generated traces by exploiting certain program characteristics and tracing only the region of execution that is relevant to the fault is proposed. Experiments indicate that the

compact representation for trace compression can help reduce the space required to store the generated traces by upto an order of magnitude and the execution reduction technique that traces only the relevant portions of the execution can reduce the size of the generated traces by 2 to 5 orders of magnitude.

For applications that are critical and for which down time is highly detrimental, techniques for surviving software failures and letting the execution continue are desired. This dissertation proposes one such technique to recover applications from a class of faults that are caused by the execution environment. The technique survives a fault by rolling back the execution to an appropriate program point and re-executing the code region under a modified environment. The environment modification (patch) that prevents the fault is noted to be re-applied if necessary to avoid the fault from recurring again in the future. This technique has been successfully used to avoid faults in a variety of applications caused due to thread scheduling, heap overflow, and malformed user requests. Case studies indicate that, for most environment bugs, the point in the execution where the environment modification is necessary can be clearly pin-pointed by using the proposed system and the fault can be avoided in the first attempt. The case studies also show that the patches needed to prevent the different faults are simple and the overhead induced by the system during the normal run of the application is less than 10 %, on average.

In a nutshell, this dissertation shows how dynamic slices can be used in fault location for multithreaded programs and addresses the challenges of scaling it for large executions. Also, it proposes techniques which let applications survive faults, caused due to the execution environment.

Chapter 1
Introduction

Software controls many systems we come across and use on a daily basis. It is used in homes, businesses, hospitals, banks, transportation and restaurants to name a few. It is also central to the success of many space missions where consequences of faults can be catastrophic. Hence, it is not surprising that software reliability is of paramount importance. For instance, the *Y2K* problem or the *millennium bug* [5] was a big concern during the turn of the century as it was thought to potentially affect critical industries like electricity and finance and its fix resulted in a world-wide expenditure of more than 300 billion dollars. It has been noted in a study by the National Institute of Standards and Technology (NIST) [18] that software related failures cost the US economy about 59.5 billion dollars annually. Hence, software reliability has been a hotbed of research and will be in the years to come given that software is only getting more complex.

Faults due to software are inevitable as the programmers are prone to mistakes. Hence, techniques to analyze the fault after it has occurred and correct the software is very important. Traces of faulty executions are collected and analyzed to isolate the root cause of the fault. This process of fault location and debugging, if done correctly, will prevent the fault from occurring in future runs of the software. Though this approach permanently removes the error from the software, there are instances when applications must survive software failures and continue execution. Critical applications cannot survive long down times, when the error is analyzed and fixed, as it can be detrimental and must continue to execute even after the fault has occurred. For instance, a software error [83] brought down an emergency management system (911) in the city of San Francisco. To make the system active the error was patched but not fixed. Hence, in these instances, techniques for fault avoidance or tolerance are important. They help in surviving the software failure and keeping the application active until the error is fixed and the software is updated.

This dissertation first shows how dynamic slices can be extended for fault location in multithreaded programs. Previous work [113] on dynamic slicing has focussed on single-threaded programs. This dissertation also shows that dynamic slices can be used to debug errors due to data races by incorporating additional dependences, resulting from the multiple threads in the program, in the dynamic slice. Constructing dynamic slices from faulty executions of long-running multithreaded programs can be challenging because they generate a lot of dynamic information which needs to be processed to form the slices. This dissertation proposes techniques to address the problem of scalability arising from these long executions. Finally, for applications that are critical and for which down-time is highly detrimental, this dissertation proposes techniques that can survive software failures and let the application continue the execution. This technique can recover the applications from a class of faults that are caused by the execution environment. The technique survives a fault by rolling back the execution to an appropriate program point and re-executing the code region under a modified environment. The environment modification (patch) that prevents the fault is noted to be re-applied if necessary to avoid the fault from recurring again in the future.

The next section is an introduction on fault location via dynamic slicing of multithreaded programs. It also identifies the challenges involved in constructing slices for long-running multithreaded program executions and provides an overview of the techniques developed for addressing the challenge. This is followed by the section which introduces the techniques developed for surviving software failures caused by the execution environment. Finally, the organization of the rest of this dissertation is presented.

1.1 Fault Location in Multithreaded Programs

In this section the problem of fault location in multithreaded programs is discussed. Multithreaded programs suffer from the different kind of faults that can occur in single-threaded programs. Additionally, in multithreaded programs, faults due to data races can also occur. These types of faults occur in multithreaded programs when one thread executes ahead of another thread due to the absence of synchronization and modifies the state of the execution inappropriately. In this section, it is shown how dynamic slicing can be extended to locate faults in multithreaded programs, including faults due to data races. There are two key challenges in achieving this objective. First, dynamic slice for fault location in single-threaded programs [113] capture data dependences of the type *Read-After-Write* (RAW) which are the only form of data dependences that occur in single-threaded programs. However, in multithreaded programs, additional inter-thread data dependences, *Write-After-Write* (WAW) and *Write-After-Read* (WAR), arise. Hence, these additional data dependences must be incorporated in the dynamic slice for faults due to data races to be analyzed. Second, In order to construct the dynamic slice of an execution, program traces have been used. However, the sizes of these traces for typical programs can easily run into the order of Gigabytes for even a few seconds of execution. Hence, techniques for scaling dynamic slicing to long-running executions is necessary.

1.1.1 Dynamic Slices of Multithreaded Programs

To ease the process of fault location, dynamic slices have been tried and tested by prior work [113]. A dynamic slice connects the various instances of executed program statements through data and control dependences such that traversing the slice from the point where the fault occurred leads to the root cause of the fault. Further, prior work has shown that dynamic slices are very effective in fault location as very few program statements need to be inspected before the root cause is spotted. For memory faults, it has been shown that it is enough to inspect less than 10 static statements to find the root cause [118].

However, prior work has mainly focussed on single threaded programs. This dissertation shows how dynamic slices can be used in multithreaded programs too. It first discusses the representation of the slice in the presence of multiple threads, showing how the various inter and intra-thread dependences (control and data) are represented. For single threaded programs, the only form of data dependences that occur are of the type *Read-After-Write* (RAW). However, for applications that are multithreaded, additional inter-thread data dependences, *Write-After-Write* (WAW) and *Write-After-Read* (WAR), arise. These additional data dependences must be incorporated in the dynamic slice for faults due to data races to be analyzed. This dissertation shows how these dependences are incorporated and presents case studies that show how the root cause of data race faults can be located using the slice when these additional dependences are present. Further, in practice, not all WAW and WAR dependences correspond to data races, and incorporating all of them in the dynamic slice leads to huge slices. Hence, this dissertation shows how the happens-before [63] algorithm can be used to restrict the set of WAW and WAR dependences captured to be only those that are potentially data races. The techniques presented can reduce the number of WAW and WAR dependences to be captured by upto 4 orders of magnitude.

The dynamic slice of a faulty execution is obtained by collecting and processing the control and data dependence information that is generated by that execution. However, for dynamic slicing to be applicable to long executions which generate a lot of dynamic information, efficient techniques are desired to collect and process this information. The next subsection discusses some of the techniques proposed in this dissertation to allow dynamic slicing to scale to long executions.

1.1.2 Scalability of Dynamic Slicing

Table 1.1. Trace sizes of program executions for small runs and their collection time in seconds.

Program	Running Time(sec)	Control Trace	Dependence Trace	Tracing Time(sec)
mysql	13	6 GB	21 GB	2886
evolution	11	87 MB	390 MB	179
balsa	17	92 MB	209 MB	1787
pftp	10	543 MB	482 MB	903
proxyc	10	1360 MB	456 MB	880
axel	8	313 MB	456 MB	184
prozilla	8	2 GB	6 GB	2640

In order to construct the dynamic slice of an execution, program traces have been used. The control- flow trace, which implicitly captures the control dependences, and the data dependence trace, are collected and processed to obtain the dynamic slice. However, the sizes of these traces for typical programs can easily run into the order of Gigabytes for even a few seconds of execution. Further, the run-time overhead that is incurred to generate these traces is significant. Table 1.1 shows the sizes of the control flow and data dependence traces and the run-time overhead for generating them. While control flow traces can be stored compactly [64], the data dependence traces are usually very large and techniques for storing them compactly do not exist. Further, for applications that are long-running like servers, it is impractical to continuously trace the entire execution. Hence, collecting and storing traces efficiently in order to perform fault location is a major challenge. This dissertation describes various techniques to perform tracing and addresses this challenge.

First, this dissertation proposes a compact trace representation called *Extended Whole Program Paths* to store on disk the control and data dependence history of a program's execution. The proposed compact trace representation is motivated by the observation that a significant fraction of the data dependence history can be recovered from the control flow trace. To capture the remainder, disambiguation checks are introduced in the program whose control flow signatures capture the result of the checks. The resulting extended control flow trace enables the recovery of otherwise irrecoverable data dependences. Experiments show that this trace can be stored very compactly in only about a third of the space required to explicitly store control-flow and data dependence traces. While this technique can be used to store the generated traces compactly, an orthogonal technique that can be used to reduce the size of the generated traces is discussed in the following paragraph.

Second, this dissertation proposes a technique to reduce the trace sizes corresponding to faulty executions in programs that are multithreaded and long-running. Since an execution of these programs can be potentially very long, it is extremely expensive in time and space to continually trace the execution online, i.e., when the application is running normally doing useful work. Also, off-line tracing, i.e., generating the traces by replaying the execution after it fails is also challenging for the following reasons. First, since such programs are non-deterministic, reproducing the failed execution off-line is non-trivial. Further, the faulty execution can generate very huge traces even if the time overhead of generating them is tolerable. The huge traces correspond to large dynamic slices which makes it very hard to construct the slices and puts a huge burden on the user who inspects these slices. In order to overcome these challenges, this dissertation proposes a framework called *Execution Reduction* in which a lightweight logging technique is used to record or log the non-deterministic events during the original execution. When a fault is encountered, the faulty execution is deterministically replayed offline using the generated event log. The proposed framework can reduce the generated trace sizes of the replayed faulty execution by tracing only the portion of the replayed execution that is relevant to the fault. Experiments show that using this framework the trace sizes can be reduced by two to five orders of magnitude.

1.2 Surviving Faults due to the Execution Environment

As mentioned earlier, there are situations in which it is important to have the application continue execution even after a fault has been observed. This dissertation proposes a technique to let applications recover from faults that occur due to the execution environment. These faults are caused by software errors that manifest under certain environmental conditions causing the execution to fail. Also, these faults can be avoided if the environment is appropriately modified. For instance, avoiding certain thread schedules in multithreaded programs can prevent certain synchronization bugs from manifesting into faults. Such faults that averted by modifying the execution environment are referred to as *environment faults*. A large number of faults that occur in today's software are environment faults. In a study by Chandra and Chen [30] and mentioned in Qin et al. [89], 56% of faults in the Apache server are dependent on the environment. This dissertation, proposes a framework to capture and recover from environment faults when they occur and to prevent them from recurring again. The faulty execution is captured by using a lightweight logging technique that records the non-deterministic events in order to allow deterministic replay. Upon a faulty execution, the code region that produced the fault is replayed repeatedly with modifications to the execution environment (e.g., changing thread schedules to avoid a synchronization fault) each time until the fault is avoided. The safe execution environment (patch) that avoided the fault is then recorded. All future executions of this application refer to the patch when executing the fault-inducing code region to try to prevent the fault from recurring again. Three different types of environment faults are considered (atomicity violation, heap buffer overflow, and malformed user request) and the system has been found to be effective in avoiding them.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 discusses in detail how dynamic slicing can be used for fault location in multithreaded programs. Chapters 3 and 4 discuss the trace representation that allows compact storage of the generated traces from a failed execution. Chapter 5 discusses the techniques that are used to reduce the size of the generated traces for long-running multithreaded programs by collecting traces for the relevant parts of the execution. Fault Avoidance is handled in Chapter 6. Related Work is discussed in Chapter 7. Chapter 8 concludes with a summary of important contributions of this dissertation and directions for future work.

Chapter 2
Fault Location using Dynamic Slicing of Multithreaded Programs

This chapter discusses how dynamic slices can be used for fault location in multithreaded programs. Previous work has shown that dynamic slices are very effective in isolating the root cause of the fault in program executions in single-threaded programs. In fact, it has been shown that for memory errors in single-threaded programs, inspecting less than 10 static statements is enough to find the root cause using dynamic slicing [118]. However, in the case of single-threaded programs the only form of data dependences that are considered is *Read-After-Write* (RAW). Whereas, in multithreaded programs additionally *Write-After-Read* (WAR) and *Write-After-Write* (WAW) must be considered if errors due to data races need to be debugged. This chapter describes the structure of the dynamic slice when these additional dependences are added and shows how to traverse the extended dynamic slice for fault location in multithreaded programs.

2.1 Dynamic Slicing of Single Threaded Programs

In this section the formal definition of a *backward dynamic slice* is provided, as defined in [113], and is shown how it is used in fault location. Let S represent the set of executed program statements for an execution. Let $s\langle t\rangle$ denote the unique execution instance of a statement s ($s \in S$) at timestamp t. A unique timestamp in increasing order is assigned to each and every statement execution instance in the order in which it is executed. Also, $s\langle t\rangle$ is said to be dependent on $s'\langle t'\rangle$ if there is a control or data dependence between the execution instance of statement s at timestamp t and the execution instance of statement s' at timestamp t'. Let CD and DD denote the set of exercised control and data dependences respectively exercised. Then, a dynamic dependence graph (DDG) is defined as follows.

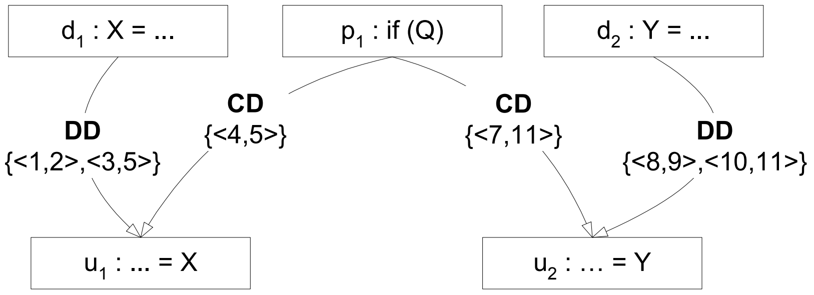


FIGURE 2.1. An example DDG for a single threaded program showing the CD and the DD edges.

Dynamic Dependence Graph. The DDG of a program's execution is a directed graph denoted as DDG (N, E) where N is the set of nodes in the graph and E is the set of edges where,

$$N = \{s | s \in S\}$$
$$E = \{(d \rightarrow u)[\langle t_1, t_2\rangle] \mid (d\langle t_1\rangle \rightarrow u\langle t_2\rangle) \in \{CD \cup DD\} \ (d \in S, u \in S)\}$$

That is, every node in the DDG corresponds to a static program statement that is executed at least once and a directed edge in a DDG is from node d to node u annotated with the set of timestamp pairs $\langle t_1, t_2\rangle$ such that execution instance of u at timestamp t_1 is dependent on execution instance of d at timestamp t_2. Figure 2.1 shows a portion of a DDG for an execution of a sample program with the control and data dependent edges. For instance, it can be seen that the fifth execution instance of statement u_1 is data dependent on the second execution instance of statement d_1 and the eleventh execution instance of statement u_2 is control dependent on the seventh instance of statement p_1.

Dynamic Slice. Given the DDG, the backward dynamic slice of this program's execution at $s\langle t\rangle$, denoted as $SLICE(s\langle t\rangle)$ is the subgraph of DDG such that $s\langle t\rangle$ is reachable by following the directed dependence edges from the source to the destination.

$$SLICE(s\langle t\rangle) = \{NSLICE(s\langle t\rangle), ESLICE(s\langle t\rangle)\}$$
$$NSLICE(s\langle t\rangle) = \{s \cup \bigcup_{\forall s' \to s\langle t', t\rangle \in E} NSLICE(s'\langle t'\rangle)\}$$

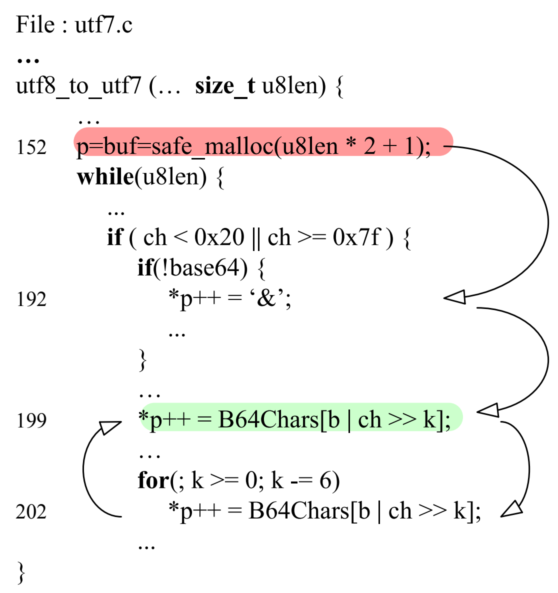
$$\text{Let } Dep.Edge = (s' \to s)(\langle t', t\rangle)$$
$$ESLICE(s\langle t\rangle) = \{Dep.Edge \cup ESLICE(s'\langle t'\rangle) \mid Dep.Edge \in E\ \}$$

That is, the backward dynamic slice of $s\langle t\rangle$ contains all statements s' such that there exists a dependence path from $s'\langle t'\rangle$ to $s\langle t\rangle$. In other words, the backward dynamic slice of an execution instance of a statement s contains all those statements upon which the execution instance of s is directly or indirectly dependent upon. Traditionally, the dynamic slice of $s\langle t\rangle$ is also defined to be the set of static program statements that influenced the value at $s\langle t\rangle$. This is nothing but the set of unique statements in $NSLICE(s\langle t\rangle)$.

During debugging, the statements and the dependence edges in the dynamic slice of the statement where the fault is observed is inspected which reveals the manner in which a bug manifested into a fault. The following subsection shows an example to illustrate this point.

2.1.1 Dynamic Slice for Fault Location in `mutt`

The program `mutt` [10] is a text based mail user agent (MUA) for Unix based Operating Systems. It has many features including customizability, POP3 and IMAP support, and ability to handle multiple mailbox formats. According to [9], mutt version 1.4 has a known memory bug which is as follows. The Mutt Mail User Agent (MUA) has support for accessing remote mailboxes through the IMAP protocol. When `mutt`



```
File : utf7.c
...
utf8_to_utf7 (... size_t u8len) {
        ...
152     p=buf=safe_malloc(u8len * 2 + 1);
        while(u8len) {
            ...
            if ( ch < 0x20 || ch >= 0x7f ) {
                if(!base64) {
192                 *p++ = '&';
                    ...
                }
                ...
199             *p++ = B64Chars[b | ch >> k];
                ...
                for(; k >= 0; k -= 6)
202                 *p++ = B64Chars[b | ch >> k];
                ...
}
```

FIGURE 2.2. A memory error in Mutt-1.4.2.1.

has to convert the name of the folder from its internal UTF-8 representation to UTF-7 it calls the function *utf8_to_utf7* in module *imap/utf7.c*. When this function does the conversion, it miscalculates the length of the output string. When a UTF-8 folder name that contains some special characters is supplied, the heap buffer overflows and a segmentation fault is flagged at line number 199.

Figure 2.2 shows the subset of static program statements that were executed, some multiple times, before the execution failed. It also shows some dependence edges without the timestamp annotations for ease of presentation. It is found that the last instance of line 199 is data dependent on line 202 and vice-versa through variable '*p*'. The arrows indicate the data dependence. Inspecting the data dependence chain in the slice, it is also found that the first instance of line 199 which is data dependent on line 192 and this in turn is data dependent on line 152, which is the root cause of the failure as there is an error in calculating the buffer length at this point. Just 8 static program statements had to be inspected before getting to the root cause, and the dependence chain provides a very clear explanation on the cause effect relations.

2.2 Dynamic Slicing of Multithreaded Programs

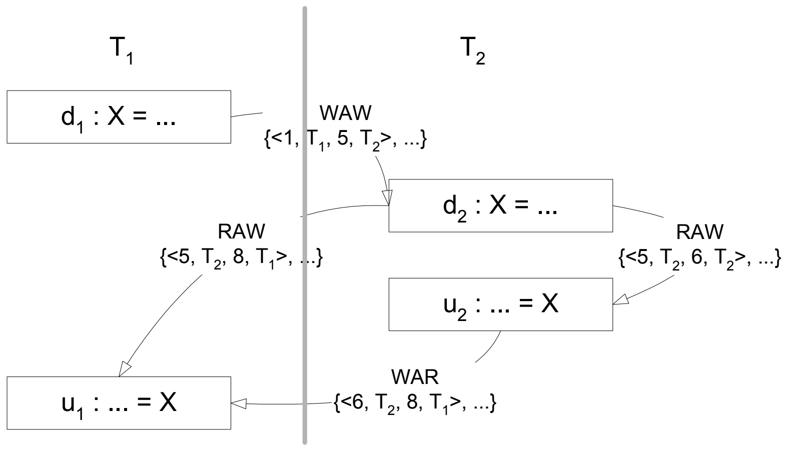


FIGURE 2.3. An example DDG showing dependence edges for a multithreaded program.

In this section, the DDG for a multithreaded program execution is discussed followed by its dynamic slice. In multithreaded programs, due to the presence of multiple threads, additional data dependences arise apart from RAW arise between the threads. They are *Write-After-Read* (WAR) and *Write-After-Write* (WAW). A WAR(WAW) dependence occurs between two threads if one thread reads(writes) a value written by another thread. Notice that while WAR and WAW can only be between different threads, RAW dependence can be within the same thread. In order to use dynamic slices in multithreaded programs for detecting data race errors, these additional dependences must also be incorporated. This section describes the DDG and the dynamic slice in the presence of these additional dependences.

Let $RACE$ be the set of dependences due to WAW and WAR in the slice. Note that CD and DD represent the set of dependences due to control and data respectively. It is assumed that the multithreaded program is running on a uniprocessor system. This gives a strict time order of the various instructions executed by the different threads and hence each instruction can be uniquely timestamped. Let N be the set of static program statements that were executed. Let $s\langle t, T\rangle$ denote the execution instance of statement $s, s \in S$ at timestamp t by thread T. The DDG can be now defined as follows. Let $(d \rightarrow u)[\langle t_1, T_1, t_2, T_2\rangle]$ denote that execution instance of u by thread T_2 at timestamp t_2 is RAW dependent on execution instance of d by thread T_1 at timestamp t_1. Let $(d \Rightarrow u)[\langle t_1, T_1, t_2, T_2\rangle]$ denote that execution instance of u by thread T_2 at timestamp t_2 is WAR or WAW dependent on execution instance of d by thread T_1 at timestamp t_1. Now, DDG (N, E) is then

$$N = \{s | s \ \in \ S\}$$

$$E = E_{ORIG} \cup E_{RACE}$$

$$E_{ORIG} = \{(d \rightarrow u)[\langle t_1, T_1, t_2, T_2\rangle] | \\ (d\langle t_1, T_1\rangle \rightarrow u\langle t_2, T_2\rangle) \ \in \ \{CD \cup DD\}\}$$

$$E_{RACE} = \{(d \Rightarrow u)[\langle t_1, T_1, t_2, T_2\rangle] \ (T_1 \neq T_2) \ | \\ (d\langle t_1, T_1\rangle \Rightarrow u\langle t_2, T_2\rangle) \ \in \ \{RACE\}\}$$

Figure 2.3 shows the dependence edges in the case of a multithreaded program. Notice that each dependence edge now additionally contains the thread ids of the statements involved in the dependences. Further, notice that while RAW edges can be inter or intra-thread, $RACE$ edges can only be inter-thread. For instance, from the figure it can be seen that there is a WAW dependence between the 1st instance of statement d_1 from thread T_1 and the 5th instance of statement d_2 of thread T_2.

The DDG is shown to consist of two types of data dependence edges, original (CD and DD) edges and the newly added WAW and WAR ($RACE$) edges. Given the DDG, the $SLICE$ can be defined as follows.

$$SLICE(s\langle t, T\rangle) = \{NSLICE(s\langle t, T\rangle), ESLICE(s\langle t, T\rangle)\}$$

$$NSLICE(s\langle t, T\rangle) = \\ \{s \cup \bigcup_{\forall s' \Rightarrow s\langle t', T', t, T\rangle \in E_{RACE}} s' \cup \bigcup_{\forall s'' \rightarrow s\langle t'', T'', t, T\rangle \in E_{ORIG}} NSLICE(s''\langle t'', T''\rangle)\}$$

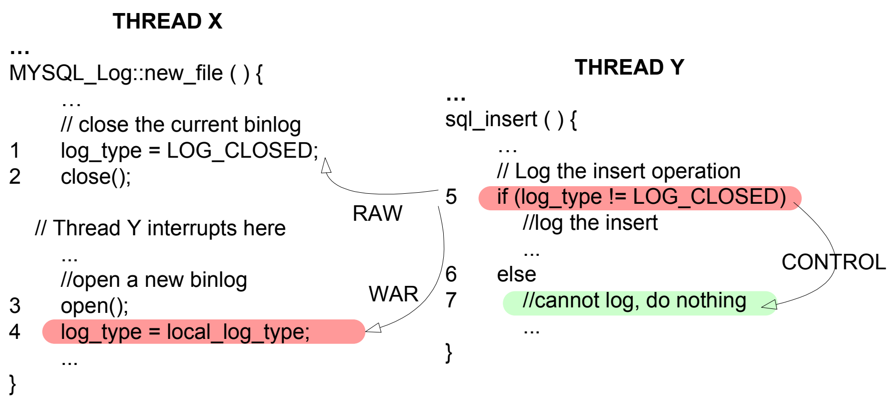
$$\text{Let } Dep.Edge_{RACE} = (s' \Rightarrow s)(\langle t', T', t, T\rangle)$$

$$\text{Let } Dep.Edge_{ORIG} = (s'' \rightarrow s)(\langle t'', T'', t, T\rangle)$$

$$ESLICE(s\langle t,T\rangle) =$$
$$\{Dep.Edge_{ORIG} \cup ESLICE(s''\langle t'',T''\rangle) \mid Dep.Edge_{ORIG} \in E_{ORIG}\} \cup$$
$$\{Dep.Edge_{RACE} \mid Dep.Edge_{RACE} \in E_{RACE}\}$$

Notice that, while computing $NSLICE$, the closure is taken over all nodes that had RAW edges but not over nodes that had WAW and WAR edges. This is because the value at the fault point could have been affected by only statements along RAW and control dependent chains. WAW and WAR edges point at places where races could have occurred but the value at the point where the fault is observed could not have been computed directly or indirectly by statement execution instances along WAW and WAR chains. The next subsection describes three examples that illustrates how a data race error is found using multithreaded slicing.

2.2.1 Dynamic Slice for Fault Location in `mysql` - I



```
...
MYSQL_Log::new_file ( ) {
        ...
        // close the current binlog
1       log_type = LOG_CLOSED;
2       close();

    // Thread Y interrupts here
        ...
        //open a new binlog
3       open();
4       log_type = local_log_type;
        ...
}
```

```
...
sql_insert ( ) {
        ...
        // Log the insert operation
5       if (log_type != LOG_CLOSED)
            //log the insert
            ...
6       else
7           //cannot log, do nothing
            ...
}
```

FIGURE 2.4. Data race error in `mysql` - I

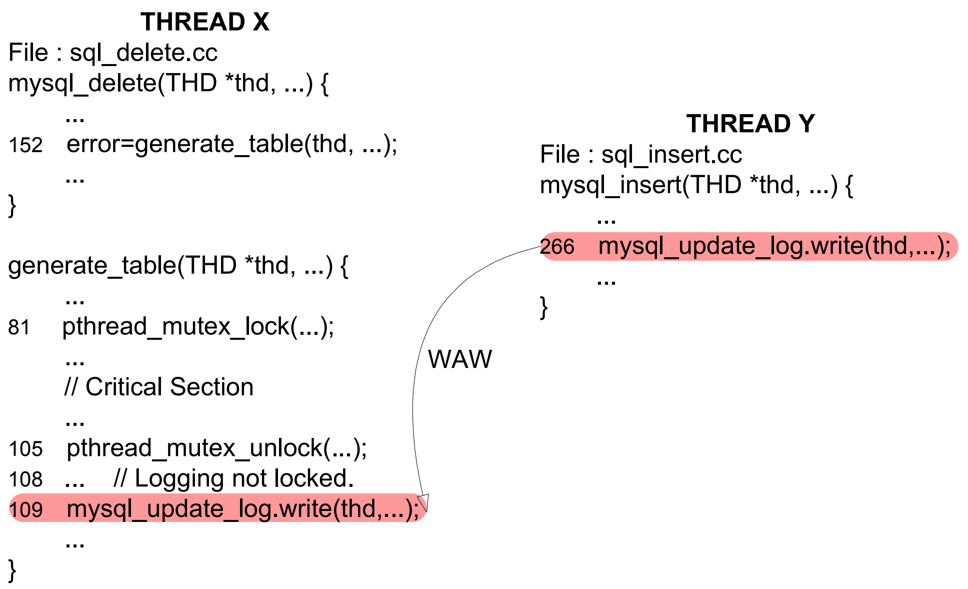
`mysql` [17] is a multithreaded application which is one of the world's most popular open source databases. It is known for its consistent fast performance, ease of use and high reliability. It is used in more than 10 million installations and runs on more than 20 platforms.

The program `mysql` ver. 4.0.12 has an atomicity violation bug [11] which is as follows. A thread that tries to close and open a new log file atomically in order to flush the previous log gets interrupted just after closing the old log by another thread that does an insert operation into a database. The second thread, hence, does not find any open log files and does not record the insert operation. These logs are used to restore databases and incorrect logs can result in inconsistency.

Figure 2.4 shows the code executed by the two threads that lead to the fault. Thread X closes the binlog (log that stores all database operations) at line 2 but before it can reopen it in lines 3 and 4, Thread Y interrupts and performs an insert operation. It tries to log the operation and checks for an open log at line 5. But, since it does not find any open log it executes the *else* part of the branch and the insert operation does not get logged. Now, once the dynamic slice is constructed and traversed from the fault point, which is line 7, the last instance of line 5 is in the slice due to the control dependence. Then, going further, it is found that the condition at line 5 obtains its value from line 1 by the RAW dependence. Notice that this value is wrong as it obtains a value of *LOG_CLOSED*. Further, the WAR dependence between lines 5 and 4 indicate the possibility of a race. Inspecting shows that this is indeed the root cause as Thread Y raced past Thread X at this point as the operations in Thread X were not locked. Less than 5 static program statements had to be inspected to nail the root cause of the error.

2.2.2 Dynamic Slice for Fault Location in `mysql` - II

According to the bug report [12], `mysql` ver. 3.23.56 has an atomicity violation error which is as follows. For some table ‘t’ in the database, when one thread does a row delete from it and another thread does an insert into it in quick succession, though the operations take place in the order they are called, they are logged in the `mysql` binlog as done in the reverse order. The `mysql` binlog does not reflect the true sequence of



```
            THREAD X
File : sql_delete.cc
mysql_delete(THD *thd, ...) {
        ...
152     error=generate_table(thd, ...);
        ...
}

generate_table(THD *thd, ...) {
        ...
81      pthread_mutex_lock(...);
        ...
        // Critical Section
        ...
105     pthread_mutex_unlock(...);
108     ...    // Logging not locked.
109     mysql_update_log.write(thd,...);
        ...
}
```

```
            THREAD Y
File : sql_insert.cc
mysql_insert(THD *thd, ...) {
        ...
266     mysql_update_log.write(thd,...);
        ...
}
```

FIGURE 2.5. Data race error in `mysql` - II

operations on the same table and hence it is inconsistent with the state of the table as shown.

—– Log File —–

SET TIMESTAMP=1151980120*;*

insert into b values (1);

SET TIMESTAMP=1151980107*;*

delete from b;

—– End of Log File —–

Notice that although the delete operation is done first it gets logged after the insert operation. The reason is that line 109 in Figure 2.5 which performs the write to the binlog is not inside the critical section. So, the thread corresponding to the insert operation gets scheduled before this point and the write to the binlog happens earlier at line 266. Now, once the dynamic slice is constructed and inspected from the fault point at line 109, the WAW dependence immediately reveals the race. Notice that this WAW dependence is through a shared file and not shared memory. Again, here less than 5 static statements had to be inspected to get to the root cause.

2.2.3 The Happens-Before Relationship

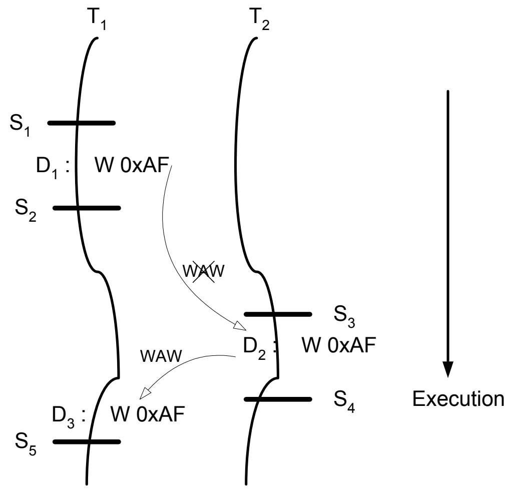


FIGURE 2.6. Sequencers in two executing threads to illustrate happens-before relation.

Potentially, the number of WAW and WAR dependences can be very large in a multithreaded program. However, not all these dependences correspond to data races. In order to restrict the dependence set to those that can be potential races, the happens-before algorithm from [63] is used. Happens-Before relationship is designed on the assumption that if shared-memory accesses are guarded appropriately using synchronizations then these cannot lead to data races. The happens-before relation provides a partial temporal order of the memory accesses dynamically based on thread synchronizations and order of execution. Now, two memory accesses from different threads that form a WAW or WAR dependence is considered for capture only if a temporal ordering, a happens-before relation, cannot be found between them. However, if there is a temporal ordering based on the happens-before relation then this dependence is not captured as this dependence is not considered a data race.

The implementation of the happens-before algorithm is done according to the procedure described in the paper by Narayanasamy et al. [79]. A sequencer (S_k), which is nothing but a global timestamp, is associated at that point of a thread's execution where a synchronization operation is executed by the thread. All the different sequencers have a strict time ordering. Any memory access M of any thread falls between two sequencers S_M and S'_M. For instance, in Figure 2.6 the write to memory location $0xAF$ falls between the interval formed by sequencers S_1 and S_2. Now two memory accesses i and j belonging to different threads that resulted in a WAW or a WAR dependence is not considered a race if their sequencer intervals do not overlap, i.e., $S'_i < S_j$ or $S'_j < S_i$. Otherwise, this dependence is a race and is captured.

Figure 2.6 illustrates this where two threads are shown to be executing with sequencers associated at points where the threads executed synchronization operations. Now, the WAW dependence between D_1 and D_2 is not captured because D_1 strictly happens-before D_2 according to the sequencer intervals encompassing them; this dependence is not considered a race. However, in the case of D_2 and D_3, where there is a WAW dependence, this is a race because the sequencer intervals which overlap do not reveal any happens-before relationship between D_3 and D_2. Hence, this dependence is considered a potential race and must be captured.

2.2.4 Capturing Relevant WAW and WAR dependences

The previous section described how to avoid capturing WAW and WAR dependences that do not result in races using happens-before relationship. In this section, it is shown how only a subset of WAW and WAR dependences need to be captured for all possible data races in the execution. Let W_1 and W_2 be two write accesses such that W_2 is dependent on W_1 via a WAW dependence. Now, this dependence is captured only if W_1 and W_2 are consecutive accesses, i.e., the W_2 is the immediate next write to that memory location following W_1. It is shown that capturing these types of dependences alone is enough to locate any data race. This is very similar to the transitive optimization [81] proposed by Netzer for replaying shared-memory programs.

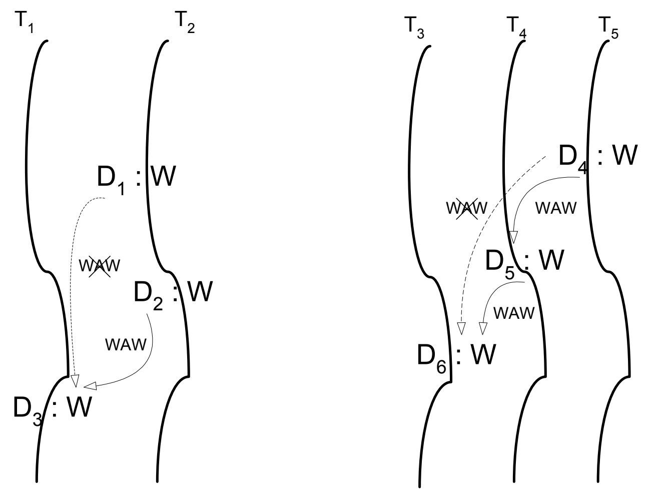


FIGURE 2.7. The relevant WAWs that need to be captured.

Figure 2.7 shows this using an example. Here, on the left two threads T_1 and T_2 are shown. Now, let there be a WAW dependence between D_1 and D_3 and also between D_2 and D_3. However, only the dependence between D_2 and D_3 is captured because only D_2 and D_3 are consecutive accesses. However, it should be noted that if the WAW between D_1 and D_3 was actually incorrect during the execution due to a race then, obviously, the WAW between D_2 and D_3 becomes incorrect too. In Figure 2.7, on the right a similar scenario is shown with 3 threads. Here, only the WAW dependences between D_4 and D_5, and D_5 and D_6 are captured. If the WAW dependence between D_4 and D_6 turned out to be a race then at least one of the two captured dependences is also a race error. The argument for WAR is similar, i.e., if a write access W_1 is WAR dependent on a read access R_1 then this dependence is captured only if the write access is the immediate next write after the read accesses. W_1 is the first write access following the read.

Table 2.1 shows the result of capturing only relevant dependences determined to be races using the happens-before relationship. The multithreaded programs are

Table 2.1. Capturing Relevant Inter-thread dependences with the happens-before algorithm (M - Million, B - Billion).

Program	Instrs.	Inter-Thread Dependences			Ratio
		Total	Relevant	Happens-Bef. (Final)	(Total/Final)
fmm	92 M	86 M	10290	4217	20394
barnes	4.3 B	81.3 M	91185	84825	958
water-spatial	1.3 B	2 M	356	150	13333
water-nsquared	1.1 B	1.6 M	244	156	10256
radiosity	907 M	2.1 B	167341	153379	13691
Average	1.5 B	454 M	53883	48545	11726

taken from the splash benchmark suite [105]. The number of threads created for each program was 4. The data shows under coulmn *Total* the total number of inter-thread WAW and WAR dependences. The column named *Relevant* shows the number of inter-thread WAW and WAR dependences that are relevant for capture as described in this section. Finally, the column labeled *Final* shows the number of relevant dependences that are races as determined by the happens-before relationship described in the previous sub-section. The data shows that the number of dependences to be captured can be reduced by upto 4 orders of magnitude.

2.2.5 Converting WAW and WAR into RAW dependences

Here, it is shown how a simple instrumentation of the multithreaded program can convert the capturing of WAW and WAR dependences into equivalent RAW dependences. As shown in Figure 2.8, every static write instruction in the program is instrumented with a read instruction to the same address immediately before it. Similarly, every static read instruction in the program is instrumented with a write to the same address, writing the same value as is read, just after it. Now, this does not affect the correctness of the program. In Figure 2.8, on the left the write at D_2

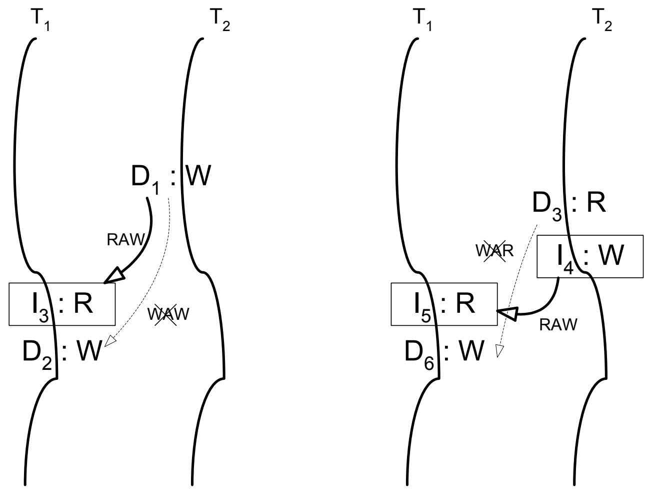


FIGURE 2.8. Converting WAW and WAR into RAW dependences.

is preceded by the instrumented read, I_3. Now, the WAW dependence between D_1 and D_2 is inferred by the RAW dependence between I_3 and D_1. There is a RAW dependence between I_3 and D_1 because D_1 and D_2 have to be consecutive memory accesses according to the previous section. Hence, the read at I_3 will get its value from D_1. On the right, I_4 and I_5 are the instrumented instructions for the read and write at D_3 and D_6 respectively. The WAR dependence between D_3 and D_6 is inferred by the RAW dependence between I_4 and I_5. The instrumented instructions are clearly marked in order to differentiate between the original RAW dependences and the synthetic RAW dependences. Notice that the ability to do this conversion is possible because all WAW and WAR dependences that need to be captured are between two memory accesses that are consecutive. Also, it should be noted that this instrumentation needs to be done only for those reads and writes that can potentially access shared memory.

Converting all dependences to be captured into RAWs helps in developing a single efficient technique rather than having to work with three different techniques for RAW, WAW and WAR. This observation is used in the rest of the dissertation.

2.3 Traces - Representing Control and Data Dependences in a File

As mentioned earlier, traces of control and data dependences are to be collected and stored in files and then later processed to build the dynamic slice of an execution. In this section, we describe how the control and data dependences are stored in a file.

The control dependences of an execution can be obtained by capturing the control-flow trace of a program's execution. The control-flow trace is nothing but the sequence of basic blocks that are executed and are stored in a file as a huge string of basic block identifiers. The data dependence trace contains the sequence of dependences exercised where each dependence is a 6-tuple of the form :

$$\langle source - stmt., Instance, ThreadId, dest - stmt., Instance, ThreadId\rangle$$

This tuple gives the information on the source and destination instruction corresponding to each exercised dependence. Alternatively, if an address trace is collected, the data dependences can then be obtained by processing the address trace. An address trace is nothing but the sequence of memory addresses accessed by load and store instructions in the program and the data dependences can be gathered by processing the address trace together with the control-flow trace. For multithreaded programs, additionally, the sequencers are also stored at exact points in the trace which correspond to the execution of synchronization instructions. Later, when the traces are analyzed, these sequencers are processed as mentioned to find happens-before relations and prune false races.

2.4 Summary

This chapter shows how dynamic slices can be used for fault location in multithreaded programs. It has also been shown that errors due to data races can be captured using the slice if the additional data dependences that arise are incorporated. It also presents some techniques to capture only the relevant WAW and WAR dependences that are races and these can reduce the number of dependences to be captured by upto 4 orders of magnitude.

Chapter 3
Compact Representation of Control and Data Dependence Traces

As mentioned before, control and dependence traces are processed to construct dynamic slices. However, the trace sizes for even small program runs can run into Gigabytes. This chapter proposes a representation that can be used to store control and data dependence traces of an execution compactly on disk. The previous chapter discussed that control flow traces and data dependence traces (explicit) or memory address traces (implicit) are collected in order to construct the dynamic slice. Actually, the traces that are collected can be represented in two possible ways: those that are more appropriate to use when the traces are *stored on disk*, such as the Sequitur [84] compressed control flow trace representation called the whole program path [64]; and those that are used when traces are *held in memory* for analysis such as the timestamped representations of control flow traces [119] and dependence traces [116]. This chapter develops a compact representation of the control and data dependence traces to be stored on disk. This representation is called the *Extended Control Flow Trace* (eCF) representation which is a *unified* representation of control flow and data dependence traces and leads to very compact trace sizes to be efficiently stored on disk.

As shown in Figure 3.1, the control and data dependence traces are collected and stored on disk. The dependence information is then *recovered* from them and the dynamic slice is formed by constructing the dynamic dependence graph.

The size of these control flow and dependence traces can be quite large for even small program runs. Table 3.1 gives an idea of the sizes of the traces for sample

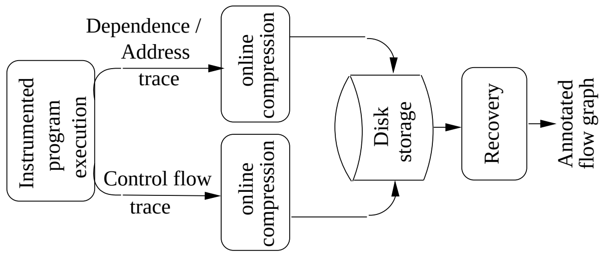


FIGURE 3.1. Trace generation, storage, and use.

TABLE 3.1. Trace sizes and compressibility.

Program	Uncomp. (MB)			Compression Factor					
				Sequitur			VPC		
	Cont.	Dep.	Addr.	Cont.	Dep.	Addr.	Cont.	Dep.	Addr.
256.bzip2	154	540	590	57	1.37	4.2	61	5.3	8.4
186.crafty	184	604	638	77	1.53	37.1	25	5.7	17.2
252.eon	115	612	812	767	1.24	1242.0	610	8.3	153.2
254.gap	72	528	593	362	1.51	2.2	179	7.2	5.93
164.gzip	197	408	564	90	1.18	5.1	116	4.5	7
181.mcf	291	687	756	1265	1.18	17.9	3417	6.2	21.5
197.parser	226	642	680	161	1.49	10.5	221	6.1	19.1
253.perlbmk	185	537	652	1542	1.20	52.4	49	4.8	8.3
300.twolf	177	513	559	59	1.25	21.0	29	4.6	7.3
255.vortex	182	618	884	3033	1.26	46.8	113	6.2	16.8
175.vpr	186	525	599	78	1.20	21.7	38	4.8	7.7
Average	179	565	666	681	1.31	132.6	442	5.8	24.8

runs. These runs were generated using the reference inputs of the **SPEC CPU 2000** integer benchmarks. The traces were collected for instruction counts of approximately between 350 and 400 million. The average length of the abridged control flow traces was around 90 million basic blocks and this corresponds to a little more than 1% of the trace of the entire run. Table 3.1 gives the sizes of the control flow traces (Cont.), and the memory dependence traces, both when the dependence is captured explicitly using dependence traces (Dep.) and implicitly using address traces (Addr.). Since the above traces were collected for the program binaries, the control flow trace implicitly captures not only the control dependences but also the register dependences. The table also shows the factors by which the traces can be compressed. It can be seen that the length of the memory dependence trace is significantly longer than the length of the control flow trace. Moreover, as shown, the compressibility of memory dependence traces, using both Sequitur [84] (grammar based compression algorithm) and VPC [28, 27] (value predictor based compression algorithm), is quite inferior to that of control flow traces. The table also clearly shows that capturing address traces is superior to capturing explicit dependence traces as they are not significantly larger and can be compressed to a greater degree. Hence, address traces in conjunction with control flow traces have been used as the baseline when evaluating the efficiency of extended control flow traces. Even though the address traces of some benchmark programs have good compressibility, overall, address traces do not get compressed as much as control flow traces. Thus, even if the address traces are compressed before being stored on disk, they can be quite long.

3.1 The Extended Control Flow Trace Representation (eCF)

In this section, the trace representation for single thread programs is presented. The next section deals with how to extend the proposed technique to multithreaded programs. As the data presented in Table 3.1 shows, control flow traces are shorter in length than dependence and address traces. This is because control flow traces consist of a sequence of executed basic blocks (or paths) while dependence traces consist of def-use information, the dynamic memory dependences, and address traces consist of the memory addresses referenced at run-time. Each execution of a basic block or path may involve several memory references. Moreover, Sequitur [84] based compression techniques are very effective for control flow traces [64] but significantly less so for dependence traces and address traces. While compression based on value predictors, VPC [28, 27], provides a greater degree of compression than Sequitur for dependence traces, this benefit comes at a price. Traces compressed using VPC have to be decompressed before they can be analyzed unlike Sequitur, which produces the compressed trace in the form of a context-free grammar that can be readily analyzed. For instance, Larus [64] has shown how to traverse the compressed control flow trace to find hot-subpaths.

The above observation served as a motivation to search for an alternative to the dependence/address trace. It should be noted that the dependence/address trace is needed because when used in conjunction with the control flow trace it enables the recovery of all dynamic memory dependences. The focus of this section is on designing an *extended control flow trace* representation from which it is possible to extract dynamically exercised memory dependences. To enable the recovery of dynamic memory dependences, the extended trace should include additional information. The following are the goals in designing the extended trace representation:

1. The additional information contained in the extended trace should be in the form of control flow so that the existing compression algorithm by Larus [64] can be used to compress the extended trace.

2. The incremental cost of generating the additional information should be minimized both in terms of the increase in the size of the trace and the increase in the program execution time due to the generation of the trace.

First, the additional information that is needed to recover the memory dependences from the control trace is discussed. Consider a path from *def*$_1$ to *use* that passes through *def*$_2$ as shown in Figure 3.2. It is assumed that memory dependences $e_1(def_1, use)$ and $e_2(def_2, use)$ are *potential* memory dependences, due to aliasing, that may or may not be manifested during a particular execution of the path. While the control flow trace will capture each execution of the path, additional information on the addresses referenced by *def*$_1$, *def*$_2$, and *use* is needed to identify the dynamic memory dependences. Thus, immediately preceding the *use*, *dynamic disambiguation checks* are introduced: *disamb* e_1 compares the addresses referenced by *def*$_1$ and *use* while *disamb* e_2 compares the addresses referenced by *def*$_2$ and *use*. As will be seen later, the control flow signature of the disambiguation checks captures the result of the comparison (true or false). Thus, the extended control flow trace (i.e., the original control flow trace augmented with the control flow signatures of the disambiguation checks) contains all the information needed to identify the dynamic memory dependences.

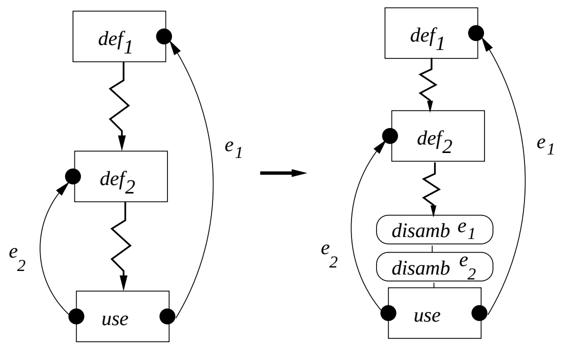
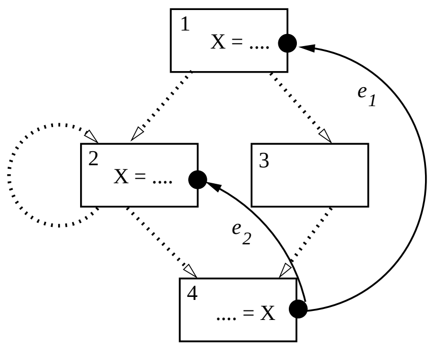


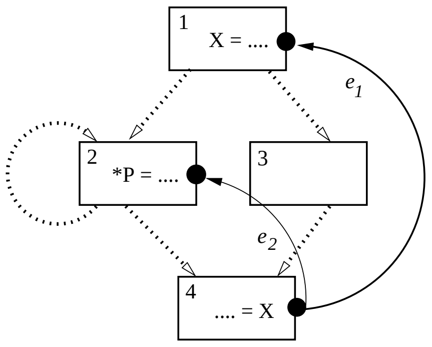
Figure 3.2. Dynamic disambiguation.

Given a set of potential memory dependences, to minimize the cost of the *disambiguation checks*, each memory dependence is classified into one of three categories: *no-cost*, *fixed-cost*, and *variable-cost* dependence. As the names suggest, the three categories differ in the cost needed for the disambiguation checks. The program transformations designed to enable this classification and the collection of the memory dependence history are described next.



(def(X:1),use(X:4))	1.3.4
(def(X:2),use(X:4))	$1.\overset{+}{2}.4$

FIGURE 3.3. Fully free.



(def(X:1),use(X:4))	1.3.4
(def(X:1),use(X:4)) or (def(X:2),use(X:4))	$1.\overset{+}{2}.4$

FIGURE 3.4. Partially free.

3.1.1 No-Cost Capture

In general, disambiguation checks need to be introduced to capture dynamic memory dependences. However, for a subset of dependences, disambiguation checks are not needed as the outcomes of these checks can be determined directly from the control flow trace.

> *Definition 1.* (Fully-free dependence) A def-use memory dependence is a *fully-free dependence* iff under every execution of the program all occurrences of the dependence can be recovered from the program's control flow trace.

Figure 3.3 illustrates this situation. The two definitions and one use in this example always refer to the same variable, i.e., X. Moreover, for path 1.3.4, it is guaranteed that dependence edge e_1 is exercised and for all other paths that arrive at 4 via 2, dependence edge e_2 is exercised. Thus, the control flow trace is sufficient to identify these dependences when exercised.

Definition 2. (Partially-free dependence) A def-use memory dependence is a *partially-free dependence* iff, in general, only some occurrences of the dependence can be recovered from the program's control flow trace.

Figure 3.4 illustrates this situation. The definition in node 2 assigns a value through a pointer. Let us assume that a *points-to* analysis indicates that the pointer P may point to variable X. For path 1.3.4, it is guaranteed that dependence edge e_1 is exercised. However, for all other paths that arrive at 4 via 2, the dependence edge e_1 may or may not be exercised. Thus, the control flow trace only captures partial information for dependence edge e_1, i.e., when exercised through 1.3.4.

The presence of free dependences can be recognized at compile time as follows. First, given a *def* that reaches a *use*, the *def* and *use* must always refer to the same variable (say X). Next, if every path from the *def* to the *use* along which the dependence can be exercised is *definition-clear* w.r.t X, then the dependence (def, use) is fully-free. If the preceding condition is only true for a subset of paths from *def* to *use* (i.e., along at least one of the paths, a definition of a *may-alias* of X is encountered), then this dependence (def, use) is partially-free.

3.1.2 Fixed-Cost Capture

Free capture is only possible when the *def* and the *use* are guaranteed to refer to the same address. If the *def* and *use* may, but not necessarily, refer to the same address, the disambiguation check must be performed at run-time. If the *def* always refers to the same variable (say X), while the *use* may or may not refer to X, introducing a *fixed cost disambiguation check* will enable detection of instances of this dependence. A fixed cost check means that every execution of the *use* will require a constant amount of additional work to perform the disambiguation check for the *def* and *use*, which is a comparison of the address of X with the address read by the *use*.

Definition 3. (Last-instance dependence) A def-use memory dependence is a *last-instance dependence* iff every occurrence of this dependence is caused by the latest execution of the definition statement prior to the execution of the use statement.

The reason why some dependences can be captured at a fixed cost is because they are *last-instance* dependences. If the *def* always refers to the same variable and if the *def* is executed multiple times prior to executing the *use*, only the last execution of the *def* is relevant to the executed *use* as the *def* assigns to the memory address every time and hence is a *last-instance* dependence.

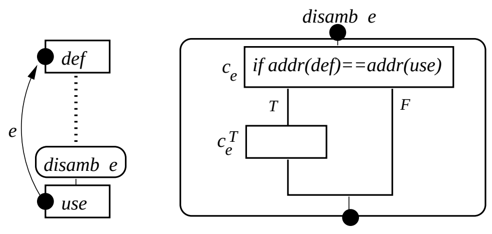


FIGURE 3.5. Fixed cost check.

A fixed-cost disambiguation check for dependence edge e, denoted as *disamb e*, has the form shown in Figure 3.5. The control flow signature of *disamb e* is $(C_e.C_e^T)$ if the check finds an address match; otherwise it is (C_e). The key point to be noted is that the result of the disambiguation check is captured by its control flow signature and is incorporated in the extended control flow trace. There is no need to explicitly save the *def* information for this *use*, i.e., the dependence trace need not be collected.

The example in Figure 3.6 illustrates a situation in which fixed-cost checks are needed to capture the three memory dependences corresponding to the use in node 5. In this example it is assumed that it is known that pointer P is not assigned in the code fragment shown. Thus the *def* in node 1 and the *use* in node 5 always refer to the same address. Assuming that P may or may not point to X or Y, disambiguation checks are needed to compare the addresses of X and Y with $*P$.

It should be noted that in the transformed program, each execution path from 1 to 5 uniquely identifies the exercised memory dependence edge. For example, consider the path 1.2.4.4.6.7.8.5. The disambiguation check signatures (6.7) and (8) indicate that P points to X not Y. The control flow 1.2 indicates that the *def* in node 2 is the latest definition of X before arriving at 5. Thus, it can be concluded that memory dependence edge e_2 is exercised along this path. Similarly, determinations can be made for all other paths.

3.1.3 Variable-Cost Capture

In the case of free-dependences, both *def* and *use* were guaranteed to always refer to the same address while in the case of fixed-cost dependences, only the *def* was always guaranteed to refer to the same address as the addresses referred to by the *use* could vary. Now, consider the final case where both the *def* and *use* can refer to varying addresses.

This final situation is illustrated by the example in Figure 3.7. When the execution proceeds along path $1.2^+.4$ (2^+ refers to one or more occurrences of 2), the value of X assigned through $*P$ in node 2 reaches the use of X in node 4. While the statements in node 2 may be executed several times, only the first execution of the definition assigns a value to X via $*P$. Thus, the dependence between the definition of $*P$ in node 2 and the use of X in node 4, denoted as $(*P : 2, X : 4)$, is not a last-instance dependence. In fact, by changing the assignment to $P = \&X$ in node 1, situations can arise where the dependence exists between *any-instance* of $*P : 2$ and $X : 4$.

> *Definition 4.* (Any-instance dependence) A def-use memory dependence is an *any-instance dependence* if an occurrence of a dependence can be caused by any one of the executions of the definition statement prior to the execution of the use statement.

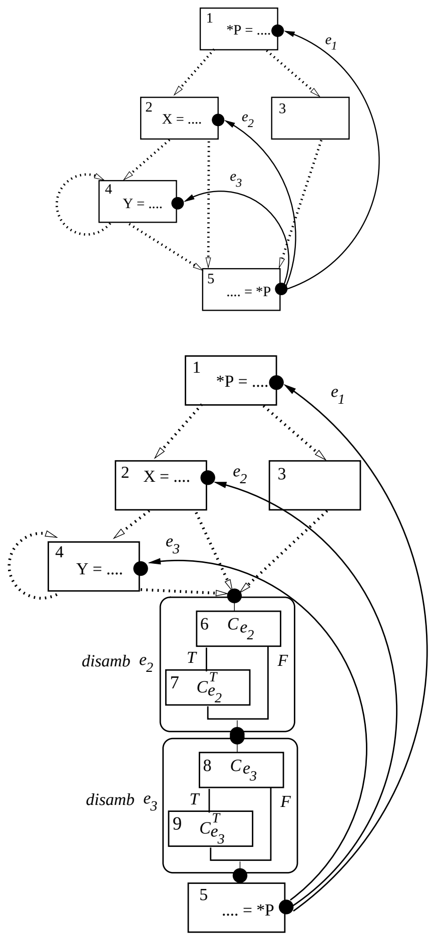


FIGURE 3.6. Fixed-cost disambiguation.

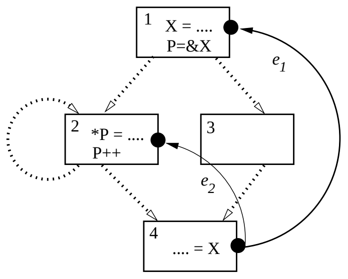


FIGURE 3.7. Any-instance dependence.

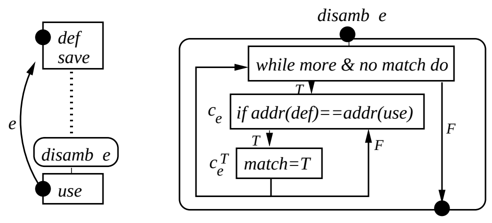


FIGURE 3.8. Variable-cost check.

To capture any-instance dependences, two things need to be done. First, all the addresses assigned to by the multiple executions of the definition must be saved in a buffer. Second, at the use, a *variable-cost* check shown in Figure 3.8 must be inserted. This check compares the *use address* with the *definition addresses* saved in the buffer one at a time starting from the latest address. The checks continue to be performed until a match is found or no more addresses remain in the buffer. The complete cost of this check is a variable as it can vary from a minimum of one check to as many checks as there are addresses in the buffer. The size of the buffer also continues to grow as the program executes. The example of Figure 3.7 once transformed using the *variable-cost* disambiguation results in the code shown in Figure 3.9.

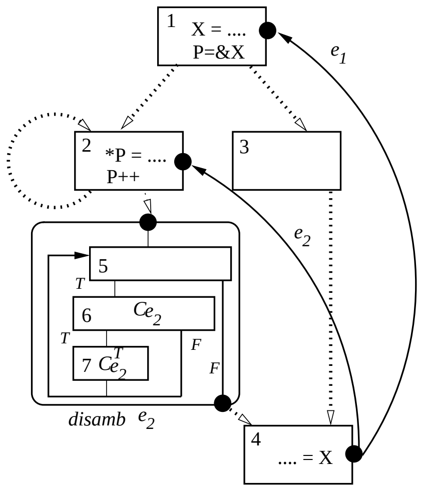


FIGURE 3.9. Variable-cost disambiguation.

3.2 Representation for Multithreaded Programs

Multithreaded executions are handled as follows. The representation described in the previous sections is used for independently representing the intra-thread control and data dependences for each thread, i.e., the extended control flow trace of each thread is stored independently. The disambiguation checks that are described are performed on each thread independently and hence they capture the intra-thread RAW dependences. Further, the thread scheduling information is also saved. This is used to put the extended control flow traces of all threads together. This is based on the assumption that the program is executing on a single processor and hence only one thread can be using the processor resources at any point of time. Once the traces of all threads can be put together, the instances of every instruction are known. Notice that the thread id of each instruction is known from the trace of the thread the instruction belongs to. Hence, the instance and the thread id of the source and the destination statement can be obtained corresponding to each dependence. Then, what remains is how to disambiguate inter-thread RAW dependences. Recall that WAW and WAR dependences need not be explicitly captured if the program is instrumented to convert all dependences to RAWs. This was explained in Chapter 2.

Disambiguation of inter-thread RAW dependences is similar to the variable-cost capture described in the previous section. It is assumed that the set of all possible inter-thread reaching definitions for every read statement have been identified using static analysis. Since, the dependence is inter-thread, the source of the dependence is treated as a pointer. Hence, the disambiguation is a search for the address in the buffer of the write instruction.

3.3 Optimizations to the Extended Control Flow Trace

The generation of the extended control flow trace involved inserting a series of disambiguation checks (instrumentation) at different points in the program. In this section a series of optimizations is presented that are aimed at tuning the insertion and execution of instrumentation code so that the size of the instrumentation code, the space and time cost of executing it, and the compressibility of the resulting trace are improved.

3.3.1 Instrumentation Code Size

Notice that disambiguation checks require computing of potential dependence information corresponding to each load. It is assumed that all potential memory dependences are identified, classified, and the program is instrumented according to the classification. However, in practice, due to the conservative nature of static analysis, too many spurious memory dependence edges may be present causing the cost of instrumentation to become very high. The unnecessary instrumentation will not only incur execution time overhead but will also increase the length of the extended control flow trace and the cost of recovering memory dependences.

To solve the above problem, a two phase profiling scheme is used that consists of a *filtering phase* and a *collection phase*. In the filtering phase, the program is instrumented to identify all memory dependence edges that are exercised at least once during execution. Also, based upon their behavior, the dependences are classified as no-cost, fixed-cost, or variable-cost. Now that all actually encountered memory dependences have been identified, the program is instrumented only with the disambiguation checks that are needed to capture these dependences. The instrumented program is then run to collect the *extended whole program path*. The instrumentation needed for the filtering phase is similar to the one used by Agrawal and Horgan [20] for their second approximate slicing algorithm – mapping between an address and the statement that defined it last is maintained to detect all exercised memory dependence edges.

This approach is not directly applicable in the presence of non-determinism since the second run on the same input may exercise some dependences that were not exercised during the first execution. The absence of instrumentation code for such dependences can cause such dependences to be missed. One solution to this problem is to conservatively introduce instrumentation code to capture all potential memory dependences. Another solution is to capture non-deterministic events in the first run and replay them using the second run so that no new dependences are exercised. For the multithreaded programs considered, non-determinism exists as the scheduling decisions need not be similar in each run. Hence, a logging scheme [96] was used to record the original execution and deterministically replay it the second time.

3.3.2 Trace Length and Compressibility

Each time a load is encountered, the disambiguation codes for all the corresponding stores (potential sources of the dependence) are executed. Therefore, the corresponding trace produced can be very long. A simple optimization can ensure that only the disambiguation code for a single store is executed. Track the last store for each address at runtime and use it to quickly identify the source of the dependence. Then, implement this by using a hash table that is indexed by the memory address and stores the identifier of the source statement that last wrote to this address. The instrumentation code for only this source is executed – the purpose of the trace produced is then to only identify the precise instance of this defining store. This optimization is shown in Figure 3.10. Note that not only the length of the trace produced is reduced but also the cost of executing the instrumentation code.

Next, consider another optimization that is aimed at sharing the instrumentation code across different uses (loads). This optimization not only reduces the overall size of the instrumentation code that is inserted but also increases the compressibility of the trace produced by this instrumentation code. A single copy of the instrumentation code as shown in Figure 3.11 is created For each load, its corresponding stores are

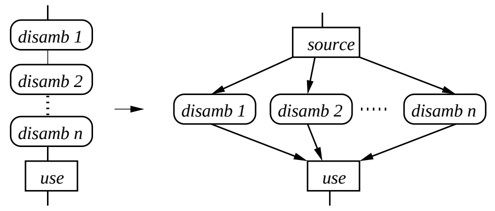


FIGURE 3.10. Optimizing trace length by executing the disambiguation code of the correct store instruction.

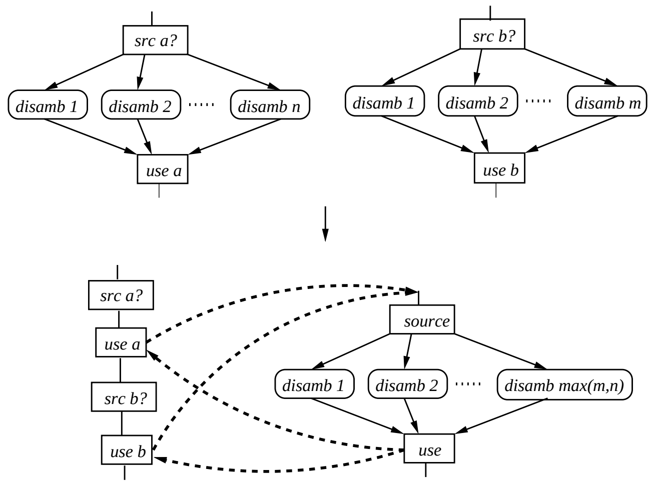


FIGURE 3.11. Optimizing trace compressibility by executing common piece of code.

numbered from 1 to n ($\leq maxn$). At each load, the source of the dependence is determined and a call is made to the shared instrumentation providing the source id ($1 \leq id \leq n$) and a pointer to its corresponding buffer. The instrumentation code is then executed producing traces such that traces for different loads now look similar thus enabling a greater degree of compression. The control flow trace produced still uniquely identifies the dynamic memory dependences. By finding the source of the call to the instrumentation code from the control flow trace, the load execution that is being processed is determined. Then, by examining the control flow trace produced by the instrumentation code itself, the source of the dependence (1 to n) and the specific execution instance of the source that is involved is known. The compressibility of the trace improves because each disambiguation involves executing a common piece of code and hence, these basic blocks repeat in the extended control flow trace. Sequitur or VPC is then able to effectively capture these repetitions and compress them.

3.3.3 Reducing the Number of Checks

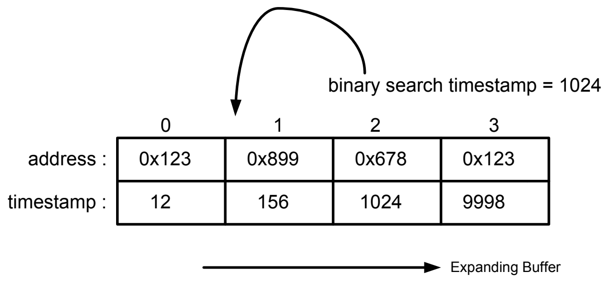


FIGURE 3.12. Reducing the number of checks.

```
BinarySearch($def_{TS}$[ ], timestamp) {
    $length = number\ of\ entries\ in\ def_{TS}$;
    $top = length, bot = 1, mid = (bot + top)/2$;
    **while** ($def_{TS}$[mid] $\neq$ timestamp) **do**
    {
        **if** ($def_{TS}$[mid] $<$ timestamp)
            $B_0 :\ top = mid - 1$;
        **else**
            $B_1 :\ bot = mid + 1$;
        $mid = (bot + top)/2$;
    }
    $E$ : **end while**
}
```

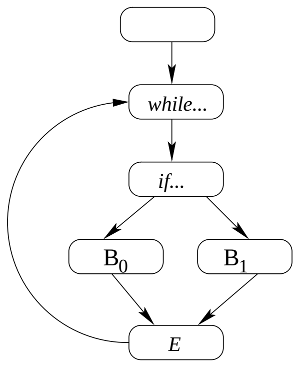


FIGURE 3.13. Code for binary search.

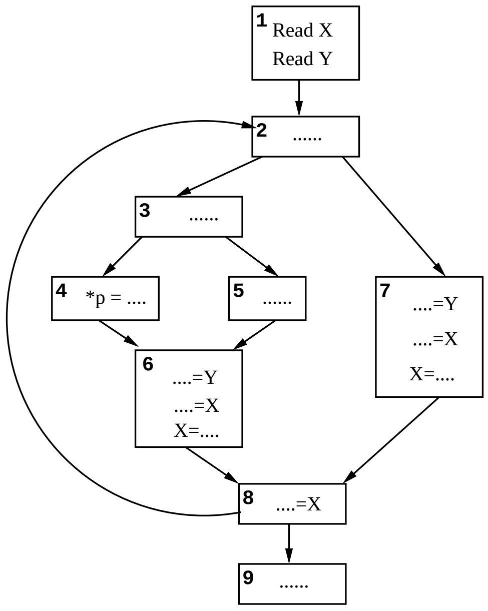
For the variable-cost transformation, the number of checks could be as high as the number of addresses stored in the buffer. This cost could significantly increase the runtime overhead. A great reduction is achieved to this cost by using the following optimization. Instead of using a linear search, the buffer is adapted to allow binary search by saving along with the address, the global timestamp at which the address was written to by the store instruction. At runtime, the global timestamp of the last write to each address is tracked. Now, at runtime, when a load is encountered, a lookup is performed to the timestamp of the latest write to the address being referenced by the load. The address in the buffer is searched using the last write timestamp. Since the timestamps in the buffer appear in ascending order, binary search is used to find the relevant timestamp and hence determine the distance. Replacing linear search by binary search greatly reduces the number of checks required. For instance, if 1 billion instructions are executed, the number of checks for each load cannot exceed $\log_2(1 \text{ billion}) = 30$. For the benchmark runs that were considered, on average, only 10 checks were needed for every dynamic dependence exercised.

Figure 3.12 illustrates the above approach. It shows a snapshot of a sample buffer. Let us say that a load corresponding to address $0x678$ is encountered. The last write information for the address implies that the timestamp at which the last write to this address was performed is 1024. Now a search is conducted for timestamp 1024 using binary search as the timestamps appear in ascending order. Once the proper entry in the buffer is found, the distance can be determined. Figure 3.13 shows the code and its CFG that does this search. def_{TS} refers to the array of timestamps, which correspond to the different instances of the definition.

The extended control flow trace will include the control flow signatures from the binary search routine that is executed to capture every variable cost dependence, implicitly capturing the definition and its instance responsible for this dependence. The next section discusses how to recover this information from the trace.

3.4 Recovering the dependence information from the traces

In this section it is described how the traces are processed when they have to be stored in memory and analyzed for use in various applications. First, how the memory or data dependences are expressed as annotations on the static program representation (similar to the dynamic dependence graph presented in the previous chapter) is discussed. Such annotated representations are very useful when these dependences have to be stored in memory for analysis and have been discussed in [119, 116].



Time Stamp (t_s)	Control Flow Trace (CF)	Dependence Trace (DD)
1	1	$\cdots$
2	2	$\cdots$
3	3	$\cdots$
4	5	$\cdots$
5	6	Y(1,1), X(1,1)
6	8	X(6,1)
7	2	$\cdots$
8	7	Y(1,1), X(6,1)
9	8	X(7,1)
10	2	$\cdots$
11	3	$\cdots$
12	4	p = &X
13	6	Y(1,1), X(4,1)
14	8	X(6,2)
15	9	$\cdots$

FIGURE 3.14. Control flow & dependence trace.

Consider the execution traces in Figure 3.14 in which the dependences are explicitly represented. The control flow trace CF gives the sequence of basic block ids executed. Let us assume that $*p$ corresponds to the contents of the address of X in this run ($p = \&X$). The dependence trace representation in Figure 3.14 is interpreted as follows. At $t_s = 14$, $X(6, 2)$ means that the use of variable X at this execution point was data dependent on the second execution instance of basic block 6. That is, the definition corresponding to the use at $t_s = 14$ comes from the second execution instance of basic block 6, which is the definition of X at $t_s = 13$. Given a *use* at some execution point, its corresponding *def*, which is the program statement and the instance, can be directly obtained from the dependence trace as the dependences are

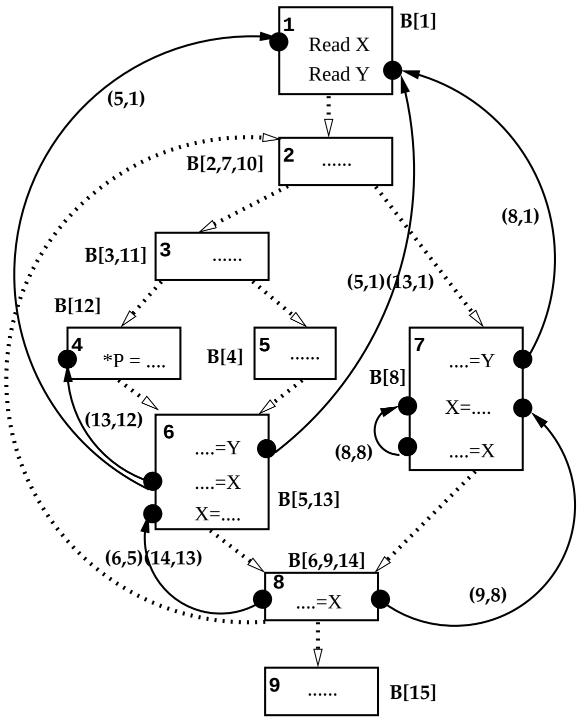
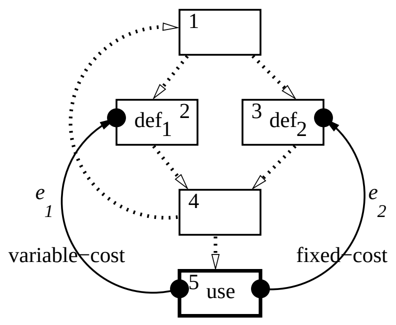


FIGURE 3.15. Annotated trace representation.

explicit. Now let us discuss how the dynamic control flow and dependences can be annotated on the static program representation. First, executions of basic blocks are assigned timestamps in the order of their execution. The column t_s of Figure 3.14 gives the timestamp values for the sample execution. Using these timestamps, the control flow trace can be annotated on the static control flow graph representation by labeling each basic block with the sequence of timestamps at which it was executed (see the timestamps prefixed by 'B' in Figure 3.15). In Figure 3.15, the control flow edges are represented by dotted lines and the dependence edges are shown in bold lines. A dynamic dependence (data or control) is annotated by labeling a static dependence edge with a sequence of timestamp pairs such that the pair of timestamps identify the execution instances of the statements that were involved in the dynamic dependence. Figure 3.15 shows the labels that identify the dynamic memory dependences next to each dependence edge. Note that the annotated representation explicitly captures the control flow and data dependences exercised in a program run.

The control flow trace and the dependence traces are explicit representations as they can be used as is. However, the address trace is an implicit representation of dependence information as it merely stores the different virtual memory addresses that were accessed during each load and store instruction. To obtain the dependence information, this address trace needs to be processed off-line. The extended control flow trace that is proposed in this dissertation is also an implicit representation which will be described in the following sections. The algorithms for recovery of dependence information from the address and extended control flow traces are described in the next subsection.

3.5 Recovering Dependence Information from eCF



Control flow and addresses referenced:
1.2(Y).4.1.3(X).4.1.2(X).4.1.2(Z).4.5(Y)

Control flow signature of disambiguation checks before 5:
$(C_{e_1}.B_0.E)$

FIGURE 3.16. Recovery example.

Given the extended control flow trace, to recover the definition corresponding to a given execution of a use, two types of information need to be put together that is contained in the control flow signatures of the disambiguation checks that immediately preceded the use. The control flow signatures of disambiguation checks that contain

```
AnnotateControlFlow( ) {
    Let $n$ be a node in the CFG
    $time = 1$;
    while not eof(trace) do
        $n = getnextnode(trace)$;
        $TS(n) = TS(n) \cup \{time\}$;
        $time{+}{+}$;
    end while
}
```

FIGURE 3.17. Annotating CFG with control flow information.

$$\forall\ (t, n) \text{ st } t \in TS(n)$$

$$Before(n(t)) = \begin{cases} \phi & if\ t = 1 \\ n'(t-1) & elseif\ (n' \in Pred(n))\ \wedge\ (t - 1 \in TS(n')) \end{cases}$$

$$After(n(t)) = \begin{cases} n'(t+1) & if\ (n' \in Succ(n))\ \wedge\ (t + 1 \in TS(n')) \\ \phi & otherwise \end{cases}$$

FIGURE 3.18. Definitions of Before and After functions to traverse the eCF along control flow edges.

the definition and the control flow of the binary search routine in Figure 3.13, which identifies the instance of the definition. Of particular importance is the ordering of the instances of the basic blocks B_0 and B_1. By putting these two pieces together, the definition and its instance that was involved in the dependence with the current instance of the use can be recovered. The algorithm to do this is discussed next.

Consider the example shown in Figure 3.16. The disambiguation code preceding the use is not shown. The trace shows that prior to reaching the use in 5, def_1 is executed three times and def_2 is executed only once. Let the control flow signature of the disambiguation checks preceding 5 be $C_{e_1}.B_0.E$. The control flow signature contains C_{e_1}, which is the signature for def_1, and hence, def_1 is the definition that produced the value. Also, look at the control flow signature of the binary search

```
RecoverDependence( $u(t)$ ) {
    Step 1: Search for the definition of the dependence
    by looking for the disambiguation check.
    Let $\{d_1,d_2 \ldots d_n\}$ be the reaching definitions of $u$.
    $n(t_n) = Before(u(t))$;
    while $n$ is not of type $C_i$
        $n(t_n) = Before(n(t_n))$;
    end while
    /* $n$ is of type $C_i \Rightarrow$ definition is of type $d_i$ */
    Step 2: Compute the instance of the definition
    by looking at the signature of binary search
    $Array A = TS(d_i)$;
    $length = count(A)$;
    if $addr(d_i) =_{definitely} addr(u)$ then
    /* $A[length]$ is the instance, last instance of $d_i$ */
        return $(d_i(A[length]), u(t))$
    $top = length, bot = 1, mid = (top + bot)/2$;
    $n(t_n) = After(n(t_n))$;
    while $n$ is not $E$
        if $n$ is $B_0$
            $top = mid - 1$;
        else
            $bot = mid + 1$;
        $mid = (top + bot)/2$;
        $n(t_n) = After(n(t_n))$;
    end while
    /* $A[mid]$ is the necessary instance of $d_i$ */
    return $(d_i(A[mid]), u(t))$;
}
```

FIGURE 3.19. Recovering the dynamic memory dependence from the extended trace for use u executed at time t by replaying binary search.

```
RecoverChain( $u_0(t_0) \rightarrow (d_1, u_1) \rightarrow \cdots (d_n, u_n) \rightarrow d_{n+1}$ ) {
  **for** $i = 0$ **to** $n$ **do**
    $(d(t), u_i(t_i))$ = RecoverDependence($u_i(t_i)$)
    **if** $d \neq d_{i+1}$ **then** return($nil$) **endif**
    $t_{i+1} = t$;
  **end for**
  return ($u_0(t_0) \rightarrow (d_1, u_1)(t_1) \rightarrow \cdots (d_n, u_n)(t_n) \rightarrow d_{n+1}(t_{n+1})$)
}
```

```
ChainFreq( $u_0(t_0) \rightarrow (d_1, u_1) \rightarrow \cdots (d_n, u_n) \rightarrow d_{n+1}$ ) {
  **for each** $t \in TS(u_0)$ **do**
    **if** RecoverChain( $u_0(t) \rightarrow (d_1, u_1) \rightarrow \cdots (d_n, u_n) \rightarrow d_{n+1}$ ) $\neq nil$
    **then** $freq^{++}$ **end if**
  **end for**
  return ($freq$)
}
```

FIGURE 3.20. Recovering a dynamic data dependence chain and obtaining its frequency.

routine (Figure 3.13) preceding the use. Now, def_1 was executed 3 times in this example and hence, the length of the timestamp array corresponding to def_1 is 3. The presence of B_0 in the disambiguation check indicates that an address match must have occurred at the first timestamp. Hence, it can be concluded that the first instance of def_1 must have been responsible for the dependence. Notice that, in extended control flow traces, the dependences are implicit. The dependence information is actually embedded in the control flow of the disambiguation checks. To recover the exact dependence, these disambiguation checks have to be interpreted in the exact reverse of the process that was used to embed them.

Next, the detailed algorithm is presented for recovering a memory dependence as illustrated by the above example. It should be recalled that the goal is to process the extended control flow trace and produce the annotations on the static program

```
RecoverDependence( $u(t)$ ) {
    Step 1: Back track and find the latest definition and its
    instance which wrote the same address as the use.
    Let the address at $u(t)$ be $addr(u)$
    $match = 0$;
    **while** $match \neq 1$
        $n(t_n) = Before(u(t))$ ;
        **if** $n$ is a definition
            **if** $addr(n) = addr(u)$
                $match = 1$;
    **end while**
    return $(n(t_n), u(t))$;
}
```

FIGURE 3.21. Recovering the dynamic memory dependence from the address trace for use u executed at time t.

representation as shown in Figure 3.15. First it is shown how the control flow trace can be traversed and annotated on the static representation. As Figure 3.17 shows, each node n is annotated with a timestamp sequence $TS(n)$. The function *getnextnode*() returns the identity of the node that was executed immediately after the current node. $TS(n)$ is the set of all timestamps at which node n was executed. *Before*() and *After*() in Figure 3.18 show how the static graph can be traversed in the reverse and forward direction of actual observed control flow using the $TS()$ annotations. Given an execution point $n(t)$, execution instance of node n at timestamp t, the function *Before*($n(t)$) returns the identity of the node(n') that was executed immediately before $n(t)$, which is $n'(t-1)$. Analogously, *After*($n(t)$) returns the identity of the node(n') that was executed immediately after $n(t)$, which is $n'(t+1)$.

Now, consider the memory dependence recovery algorithm in Figure 3.19. The first step in the algorithm examines the control flow signature of the disambiguation checks before the use to determine the definition that was involved in the dependence. This is done by traversing backward from the use point, $u(t)$, till a node whose signature is of the form C_i is found. The signature of this node exactly gives the definition, d_i, of this use. Once this is obtained, the second step looks at the control flow signature of the binary search routine to determine the exact instance of the definition that produced the value referenced by $u(t)$. To use this signature, the array of timestamp values that was used to embed this signature has to be reconstructed. The array of timestamp values is already available in $TS(d_i)$. This array contains the timestamp values of execution instances of the definition d_i. Using the control flow signature of the binary search routine, this array is searched to obtain the right instance. The control flow signature gives the binary search order that was used on the timestamp array to embed the dependence and retracing this will give us the exact instance. When the definition and its instance are found, the memory dependence is recovered. Note that this process is the exact reverse of the process used to embed the dependence. All memory dependences can be recovered and annotated on the program representation in this manner.

While a single call to the RecoverDependence recovers a single dynamic memory dependence, additional functions such as RecoverChain and ChainFreq can be developed by building upon this that are able to recover a *dependence chain* and the number of times a dependence chain is encountered during execution (see Figure 3.20). In Figure 3.20, d_1 is the definition corresponding to the use $u_0(t_0)$. u_1 is a use at statement d_1 and d_2 is the definition corresponding to use u_1 and so on. The RecoverDependence routine is used to check if a particular chain has been executed.

The algorithm for recovering a memory dependence using an address trace is shown in Figure 3.21. At every use point, a backtracking is done to find the latest definition that wrote to the same address as the use. Figure 3.21 shows a simple backtracking scheme. However, the backtracking can be implemented more efficiently by using a hash table as follows. For every definition encountered, the instance and the definition in the hash table indexed by the address is stored. When a use is encountered, a look up of the hash table will retrieve the latest definition and instance that wrote to the same address.

3.6 Summary

In this chapter, the extended control flow trace representation which implicitly captures control and data dependences was presented. After discussing how traces will be stored in memory for analysis, algorithms for processing the extended control flow trace to recover the control flow and data dependence information was presented. The next chapter presents compression schemes used to make the traces more compact before being stored on disk and a detailed experimental evaluation of the developed techniques.

Chapter 4
Compression of Control and Data Dependence Traces

In the previous chapter, the design of the extended control flow trace was discussed. Recall that the extended control flow trace representation is designed for producing compact dependence (control and data) traces that can be stored on disk efficiently. This chapter discusses compression of the extended control flow traces (eCF) which enables these traces to be stored very efficiently. The compressed eCF is referred to as the Extended Whole Program Path (eWPP) analogous to the Whole Program Path (WPP) [64] which is the compressed representation of control flow traces. The compression of these traces is tested independently with two known algorithms; Sequitur [84], a context-free grammar based compression algorithm, and VPC [28, 27], a value predictor based compression algorithm, that are known to compress program traces well. Also, this chapter presents detailed experimental evaluations confirming the benefit of using the extended control flow trace representation for storing traces on disk.

4.1 Overview of Compression Schemes

The compression schemes that have been used to compress the traces is discussed in this section. Two compression algorithms have been used independently to evaluate the compressibility of the different traces. VPC [28, 27] is a value predictor based compression scheme and Sequitur [84] is a context-free grammar based compression scheme. This section provides a brief overview of both the schemes. Further, an enhancement to the Sequitur compression algorithm is proposed that improves its compressibility.

4.1.1 Overview of VPC

VPC is a value prediction based compression algorithm that exploits repetitive patterns in the input trace to compress it. It is a one-pass algorithm and runs in time that is linear in the length of the input trace. The algorithm compresses the trace in the following manner. At any particular point in the input trace, after having inspected a certain number of input symbols, the algorithm predicts the next symbol based on the current history. If the prediction is correct then the symbol is stored using just one bit, say '1'. If there is a mis-prediction, a '0' bit is stored to indicate that followed by which the entire input symbol is stored. Hence, the algorithm works well if the input has repeating patterns that increase the predictability of each symbol. The predictors themselves are based on Finite Context Method (FCM) prediction. Decompression of the compressed traces is analogous. If the bit read from the compressed trace is a '0' then the input symbol is also read from the compressed trace, otherwise the predicted value is used.

4.1.2 Overview of Sequitur

Sequitur is a context-free grammar based compression algorithm that exploits repetitions in the input string to compress it and runs in time linear in the length of the input string [84]. As an example, for the input string

abcabcabc

Sequitur produces the following grammar.

$$S \rightarrow AAA$$
$$A \rightarrow abc$$

In this example, Sequitur is effectively able to capture the repetition of the pattern abc. The Sequitur algorithm manipulates the input symbols so that the following two properties are preserved.

1. *Digram Uniqueness.* A digram is a pair of symbols that occur together in the input string. This property states that any digram "xy" can occur at most once in the entire grammar. If it occurs more than once, Sequitur introduces a new rule, of the form {R → xy}, that replaces both occurrences of the digram "xy" with the left hand side of the rule, 'R'. For example, after abca has been processed in the example discussed above, the next symbol causes digram ab to occur twice. So, Sequitur produces a new rule $A \rightarrow ab$ and transforms the input into AcA.

2. *Rule Plurality.* This property states that the left hand side of any rule must appear more than once in the entire grammar, on the right hand side of the other grammar rules. If any rule occurs only once then Sequitur deletes this rule and substitutes the rule occurrence with its right hand side. In the example above, after abcabc is processed, the resulting grammar rules are $S \rightarrow BB$, $B \rightarrow Ac$, and $A \rightarrow ab$. Now, rule A occurs only once on the right hand side of the entire grammar. Hence, to preserve this property, Sequitur expands rule $B \rightarrow Ac$ into $B \rightarrow abc$ and deletes the rule $A \rightarrow ab$.

4.1.3 Enhancement of Sequitur

The motivation to enhance Sequitur occurred by observing the way it compresses repeating digrams. To illustrate this with an example, consider the input string

wbcwbcxbcxbcybcybczbczbc

Using Sequitur to compress this produces the grammar :

$$S \rightarrow AABBCCDD$$

$$A \rightarrow wbc$$

$$B \rightarrow xbc$$

$$C \rightarrow ybc$$

Table 4.1. Compression Ratio, Memory Used and Runtime Overhead : Original Sequitur versus Enhanced Sequitur

Program	Compression Ratio		Memory Used		Runtime
	Original	Enhanced	Original	Enhanced	Enh./Orig.
256.bzip2	57	65	48 M	308 M	4.9
186.crafty	77	126	50 M	368 M	4.8
252.eon	767	1323	22 M	230 M	4.9
254.gap	362	1051	23 M	144 M	4.9
164.gzip	90	102	43 M	394 M	4.0
181.mcf	1265	1572	23 M	582 M	3.8
197.parser	161	297	36 M	452 M	4.5
253.perlbmk	1542	2728	22 M	370 M	4.9
300.twolf	59	73	54 M	354 M	4.8
255.vortex	3033	4100	21 M	364 M	3.4
175.vpr	78	93	46 M	372 M	4.8
Average	681	1048	35 M	358 M	4.5

$$D \rightarrow zbc$$

Although Sequitur detected patterns like “wbc”, it missed out the pattern “bc” that got repeated in each of the rules. A better grammar would have been :

$$S \rightarrow AABBCCDD$$

$$A \rightarrow wE$$

$$B \rightarrow xE$$

$$C \rightarrow yE$$

$$D \rightarrow zE$$

$$E \rightarrow bc$$

Notice that the above grammar contains fewer symbols than the first one. The magnitude of this saving grows with the size of the input string.

The reason why Sequitur failed to detect the pattern “bc” is because it first detected the digram “wb”, which killed the digram “bc”. Had this string been processed offline, instead of an online left to right method that Sequitur uses, and first

```
Enhanced_Sequitur( $Trace$ ) {
    Step 1: Preprocess Trace to find frequency of all digrams
    Let $T$ be the set of all ordered pairs (digram,frequency)
    sorted in descending order of frequency counts.
    Step 2: Compress the Trace
    while $T$ is not empty
    $\rightarrow$ Pick the most frequent digram from $T$, say "$xy$".
    $\rightarrow$ Create a new rule $A_i \rightarrow xy$.
    $\rightarrow$ Substitute all occurrences of "$xy$" with $A_i$.
    $\rightarrow$ Update the frequency counts of digrams in T. Add new
        ordered pairs for the newly created digrams. Delete
        ordered pair for digram "$xy$".
    $\rightarrow$ If any rule $R_i$ violates the Sequitur property of rule
        plurality, expand the rule.
    end while
}
```

FIGURE 4.1. Enhanced Sequitur Algorithm

compressed digram "bc", then the ideal grammar for this string could have been generated. Based upon this observation is the following proposal for the enhancement to Sequitur. It is first shown how better grammars can be generated by processing the string offline. Then, it is easy to see that the offline algorithm could be made online by buffering the input. Depending on the size of the buffer, a degree of approximation will however be introduced. In the experiments conducted, the compression results were reported by processing the string offline. Note that offline processing is equivalent to having a buffer whose size is the same as the size of the uncompressed trace.

The algorithm is described now. First, the entire string is scanned and find frequency counts of each digram occurring in the string is determined. Then digrams are compressed in the descending order of their frequency. As and when a digram is substituted by a rule, the two properties of Sequitur are checked to make sure they are satisfied. In this process, digrams that are more frequent are substituted first rather than the earliest occurring digrams. This ensures that highly frequent digrams are not destroyed in the process of compressing infrequent digrams and hence promises smaller grammars. For instance, in the example above, digram bc is the most frequent. So, compressing it first gives rise to the grammar

$$S \rightarrow wExEyEzE$$
$$E \rightarrow bc$$

and further compression yields the ideal grammar for this string already shown previously. The original Sequitur algorithm was modified to take into account these changes. Figure 4.1 shows the pseudocode for the new Sequitur algorithm.

Experimenting with the enhanced Sequitur found that it is very effective on control flow traces. Table 4.1 compares the compression ratios obtained by both the Sequitur versions. On average, enhanced Sequitur can compress the traces further by around 33%. Table 4.1 also compares the memory used by both versions of Sequitur and the runtime overhead of using enhanced Sequitur. The enhanced version uses ten times more memory and is five times slower, on average.

4.2 Experiments

The various techniques described for the generation and collection of the traces were developed using the *Phoenix* Compiler Framework developed by *Microsoft*. The instrumentation code was inserted by using Phoenix to rewrite the binaries of the benchmark programs. The intermediate representation which was used was the low-level *x86* instruction set. This allowed to clearly distinguish between register dependences and memory dependences. Notice that the register dependences can always be detected directly from the control flow trace. Hence, the instrumentation was performed to capture only data (memory) dependences. This is important for carrying out a realistic evaluation as for the program runs used in the experiments, on average, 77.5 percent of all dependences were register dependences for program runs that execute in

Table 4.2. Register vs. Memory dependences.

Program	Instructions (millions)	Register (%)	Memory (%)
256.bzip2	402	78.0 %	22.0 %
186.crafty	459	79.7 %	20.3 %
252.eon	378	72.0 %	28.0 %
254.gap	425	82.9 %	17.1 %
164.gzip	423	83.8 %	16.2 %
181.mcf	429	71.0 %	29.0 %
197.parser	415	74.2 %	25.8 %
253.perlbmk	354	72.2 %	27.8 %
300.twolf	405	79.2 %	20.8 %
255.vortex	418	69.4 %	30.6 %
175.vpr	407	77.5 %	22.5 %
Multithreaded Programs			
fmm	92	75.5 %	24.5 %
barnes	100	91.5 %	8.5 %
Average	361	77.5 %	22.5 %

the order of hundreds of millions of instructions (see Table 4.2). The SPEC CPU2000 C benchmarks were used to carry out the experiments (`176.gcc` was excluded because the version of Phoenix that was used could not handle this benchmark). Two multithreaded programs, `fmm` and `barnes`, were used from the *splash-2* [105] benchmark suite. Three threads were created during the execution of each multithreaded program. Since the multithreaded programs could not be handled by Phoenix, the valgrind dynamic instrumentation [80] tool was used. Two instrumented versions of each binary were created apart from the original. The first version captured control flow traces and address traces. The second version captured extended control flow traces. Being able to produce the instrumented binaries of each of these using Phoenix made it possible to accurately measure the overheads involved in collecting these traces. The binaries were executed on a system with a 2 GHz Intel processor, 2 GB of RAM and 100 GB of hard disk space. Based upon this implementation, an

Table 4.3. Uncompressed trace sizes.

Program	CF+AT (MB)	eCF (MB)	eCF/ CF+AT
256.bzip2	154 + 590 = 744	380	0.51
186.crafty	184 + 638 = 822	392	0.48
252.eon	115 + 812 = 927	414	0.45
254.gap	72 + 593 = 665	411	0.62
164.gzip	197 + 564 = 761	288	0.38
181.mcf	291 + 756 =1047	735	0.70
197.parser	226 + 680 = 906	609	0.67
253.perlbmk	185 + 652 = 837	466	0.56
300.twolf	177 + 559 = 736	417	0.57
255.vortex	182 + 884 =1066	500	0.47
175.vpr	186 + 599 = 785	318	0.41
Multithreaded Programs			
fmm	55 + 226 = 281	166	0.59
barnes	21 + 113 = 134	61	0.46
Average	157 + 590 = 747	397	0.53

experimental evaluation whose results are described next.

4.2.1 Trace Sizes

In this section, the sizes of the uncompressed and compressed traces are compared. In Table 4.3, the sizes of the *uncompressed* control flow (CF), address (AT), and extended control flow (eCF) traces are given. The traces were collected, on average, for the first 400 million instructions for the SPEC programs and around 100 million instructions for the two splash-2 programs. The corresponding *compressed* trace sizes, i.e., WPP, cDD, and $eWPP$ respectively, are also given in Tables 4.4 and 4.5. It can be seen that, on average, the eCF is smaller than $CF + AT$ by 47 percent while $eWPP$ is smaller than $WPP + cDD$ by 76 percent and 70 percent using Sequitur and VPC, respectively. In other words, in the cases of both uncompressed traces or

Table 4.4. Sequitur compressed trace sizes.

Program	WPP+cAT (MB)	eWPP (MB)	eWPP/ WPP+cAT
256.bzip2	2.4 + 142 = 144	46.8	0.32
186.crafty	1.5 + 17.2 = 19	11.6	0.60
252.eon	**0.1 + 0.65 = 1**	**6.0**	**6.00**
254.gap	0.1 + 270 = 270	2.2	0.01
164.gzip	2.0 + 110.5= 113	28.6	0.25
181.mcf	0.1 + 42.2 = 42	16.7	0.40
197.parser	0.8 + 64.8 = 66	21.0	0.32
253.perlbmk	0.1 + 12.4 = 13	1.5	0.12
300.twolf	2.4 + 26.6 = 29	32.0	1.10
255.vortex	0.04 + 18.9 = 19	4.5	0.24
175.vpr	2.0 + 27.6 = 30	17.4	0.60
Multithreaded Programs			
fmm	0.1 + 2.5 = 2.6	0.5	0.2
barnes	0.2 + 4.3 = 4.5	2	0.45
Average (excluding 252.eon)	0.9 + 62 = 63	15	0.24

compressed traces, the extended control flow trace is superior to combined control flow and address trace. For `252.eon`, using Sequitur, the eWPP trace size obtained is larger, though not significantly. This aberration is due to the fact that the address trace for this program is highly compressible using Sequitur. The average size of the eWPP is calculated excluding `252.eon`.

From the data in Table 4.4 it can be seen that the reduced size of $eWPP$ is due to two factors. First, the size of eCF is smaller than the size of $CF + AT$, as a result of the novel representation of dependences. Second, Sequitur and VPC are extremely effective in compressing eCF into $eWPP$. Now, the contribution of each of the two factors mentioned in reducing the trace size from $CF + AT$ to $eWPP$ is discussed. Table 4.6 shows the results of this experiment. It shows that, on average, for Sequitur compressed traces, 48 % of the reduction in trace size comes

TABLE 4.5. VPC compressed trace sizes.

Program	WPP+cAT (MB)	eWPP (MB)	eWPP/ WPP+cAT
256.bzip2	2.5 + 69.9 = 72	30.3	0.42
186.crafty	7.2 + 37 = 44	18.6	0.43
252.eon	0.19 + 5.3 = 5	2.1	0.42
254.gap	0.4 + 100 = 100	9.3	0.10
164.gzip	1.7 + 80.1 = 82	19.4	0.24
181.mcf	0.09 + 35.2 = 35	7.7	0.22
197.parser	1 + 35.6 = 37	20.9	0.56
253.perlbmk	3.8 + 78.3 = 82	18.4	0.22
300.twolf	6.1 + 76.4 = 83	39.4	0.47
255.vortex	1.6 + 52.7 = 54	12.2	0.23
175.vpr	4.9 + 77.8 = 83	24.8	0.30
Multithreaded Programs			
fmm	0.1 + 5.7 = 6	0.6	0.10
barnes	3 + 16 = 19	2.5	0.13
Average	2.5 + 52 = 55	16	0.30

from the first factor (shown under column *Smaller eCF*), i.e., going from $CF + AT$ to eCF. The remaining 52 % reduction comes from the compression (shown under column *Compression of eCF*), due to going from eCF to $eWPP$. For VPC too, the contributions due to both factors were 49 % and 51 % respectively. This shows that both the factors, representing the trace as eCF and then compressing it, are important in achieving smaller $eWPPs$.

Also the distribution of three types of dynamic memory dependences: no-cost, fixed-cost, and varying-cost was studied. The resulting data is given in Table 4.7. From this data it can be seen that, on average, 69.9 percent of the dependences are hard dependences, i.e., varying-cost dependences. However, the number of no-cost memory dependences is also significant (average of 26.7 percent), which contributes directly towards reducing the size of eCF.

TABLE 4.6. Reason for reduced *eWPP* size when compared to Address Trace.

Program	Sequitur		VPC	
	Smaller *eCF* (%)	Comp. of *eCF* (%)	Smaller *eCF* (%)	Comp. of *eCF* (%)
256.bzip2	52 %	48 %	51 %	49 %
186.crafty	53 %	47 %	54 %	46 %
252.eon	56 %	44 %	55 %	45 %
254.gap	38 %	62 %	39 %	61 %
164.gzip	65 %	35 %	63 %	37 %
181.mcf	30 %	70 %	30 %	70 %
197.parser	34 %	66 %	34 %	66 %
253.perlbmk	44 %	56 %	45 %	55 %
300.twolf	45 %	55 %	46 %	54 %
255.vortex	53 %	47 %	53 %	47 %
175.vpr	61 %	39 %	61 %	39 %
Multithreaded Programs				
fmm	41 %	59 %	40 %	60 %
barnes	57 %	43 %	56 %	44 %
Average	48 %	52 %	49 %	51 %

4.2.2 Runtime Overhead in Trace Collection

The execution time cost of the disambiguation checks is mainly due to the address comparisons performed. In particular, the greater the number of such comparisons, the greater is the overhead. In Table 4.8, the average number of comparisons performed per dynamic data dependence is shown in the column named binary under *Checks/Dep*. These results were obtained by applying the optimizations described in Section 4. However, the most significant factor in keeping the number of checks small is using binary search, described in Chapter 2, instead of linear search. If these optimizations had not been performed then the number of checks needed at run-time would have gone up by a significant amount, as shown in the column named linear under *Checks/Dep.*, making collection of these traces impractical.

TABLE 4.7. Distribution of memory dependence types.

Program	No-Cost (%)	Fixed (%)	Varying (%)
256.bzip2	40.8 %	3.1 %	56.1 %
186.crafty	48.5 %	0.0 %	51.5 %
252.eon	18.8 %	16.7 %	64.5 %
254.gap	3.4 %	0.6 %	96.0 %
164.gzip	72 %	0.1 %	27.9 %
181.mcf	6.9 %	3.5 %	89.6 %
197.parser	9.8 %	5.6 %	84.6 %
253.perlbmk	21.3 %	0.7 %	78.0 %
300.twolf	29.4 %	8.2 %	63.4 %
255.vortex	22.3 %	1.5 %	76.2 %
175.vpr	60.9 %	3.0 %	36.1 %
Multithreaded Programs			
fmm	1 %	0.04 %	99 %
barnes	11.3 %	1 %	87.7 %
Average	26.7 %	3.4 %	69.9 %

Table 4.9 shows the run-time overhead needed to collect these traces. The running time of the 3 versions of each program, that is, the original version, the instrumented version for collecting control flow and address traces (V_{AT}), and the instrumented version for collecting extended traces (V_E) is shown. For V_E, the time spent in the filtering phase alone is shown as FP. Also, for versions V_{AT} and V_E, the time spent on processing (CPU) and IO are separately shown. The CPU time spent in V_E is higher than V_{AT}, coming from the checks needed per dependence. The numbers also show the overhead incurred in the filtering phase (FP). On average, there is a 5$\times$ increase in runtime overhead when collecting extended control flow traces when compared to collecting control flow and address traces.

Table 4.8. Address comparisons per dep. edge using linear and binary search.

Program	Checks/ Dep.		Min	Max
	Linear	Binary		
256.bzip2	164814	11	1	24
186.crafty	18004	5	1	21
252.eon	35738	9	1	22
254.gap	661199	12	1	23
164.gzip	80493	8	1	22
181.mcf	194896	4	1	22
197.parser	107898	12	1	22
253.perlbmk	33341	8	1	23
300.twolf	170999	18	1	22
255.vortex	158386	10	1	23
175.vpr	26126	9	1	22
Multithreaded Programs				
fmm	754500	8	1	23
barnes	138900	9	1	22
Average	195791	10	1	22

4.2.3 Dependence Edge Recovery

Table 4.10 shows the time needed to recover the dependence information from the address traces and extended traces. As mentioned before, dependences in the extended control flow traces and address traces are implicitly represented. To be used in memory for analysis, they need to be recovered and the numbers show the time needed to do the same. Notice that it is much harder to recover the dependences from the address traces. Although address traces are quicker to generate they need, on average, ten times the time needed to process extended traces for recovering the dependences.

Table 4.9. Running time of instrumented versions in seconds.

Program	Original	CF + AT (V_{AT})	eCF (V_E)
	CPU	CPU + IO	FP + CPU + IO
256.bzip2	5	26 + 22 = 48	118 + 160 + 69 = 347
186.crafty	5	28 + 29 = 57	94 + 96 + 21 = 211
252.eon	3	28 + 40 = 68	105 + 113 + 51 = 269
254.gap	3	19 + 27 = 46	170 + 214 + 55 = 439
164.gzip	5	31 + 15 = 46	65 + 71 + 16 = 152
181.mcf	7	30 + 35 = 65	116 + 135 + 21 = 272
197.parser	7	28 + 32 = 60	90 + 100 + 66 = 256
253.perlbmk	5	28 + 35 = 63	90 + 217 + 55 = 362
300.twolf	6	29 + 29 = 58	77 + 87 + 54 = 218
255.vortex	4	32 + 28 = 60	139 + 143 + 48 = 330
175.vpr	7	27 + 30 = 57	86 + 95 + 51 = 232
Multithreaded Programs			
fmm	3	28 + 8 = 36	73 + 130 + 18 = 221
barnes	3	32 + 5 = 37	57 + 118 + 11 = 186
Average	5	28 + 26 = 54	99 + 131 + 41 = 271

4.2.4 Decompressing the Traces

While the traces are compressed to be stored compactly on disk, in order to use the trace information in analysis, they need to be decompressed. In Table 4.11, the time taken to decompress compressed extended control flow traces and address traces is shown. On average, Sequitur compressed traces take longer, more than twice the time, to decompress than VPC compressed traces. The total time needed to recover the dependences explicitly from compressed traces is the sum of the decompression time and the time needed to recover the dependences from the uncompressed traces shown in Table 4.10. Although there is a $5\times$ increase in runtime overhead in collecting extended traces over collecting control flow and address traces, the time taken to collect, decompress and recover the traces is of the same order ($271 + 17 + 55 = 343$ for extended traces and $54 + 157 + 70 = 281$ for control flow and address traces).

TABLE 4.10. Dependence edge recovery time in seconds.

Program	CF + AT (R_{AT})	eCF (R_E)
	CPU + IO	CPU + IO
256.bzip2	146 + 5 = 151	18 + 4 = 22
186.crafty	164 + 4 = 168	11 + 5 = 16
252.eon	206 + 5 = 211	16 + 4 = 20
254.gap	212 + 5 = 217	14 + 4 = 18
164.gzip	75 + 4 = 79	15 + 4 = 19
181.mcf	157 + 6 = 163	21 + 4 = 25
197.parser	146 + 4 = 150	16 + 5 = 21
253.perlbmk	241 + 6 = 247	8 + 4 = 12
300.twolf	165 + 5 = 170	9 + 4 = 13
255.vortex	211 + 6 = 217	16 + 4 = 20
175.vpr	141 + 5 = 146	12 + 4 = 16
Multithreaded Programs		
fmm	63 + 3 = 138	7 + 3 = 30
barnes	50 + 2 = 144	6 + 3 = 26
Average	152 + 5 = 157	13 + 4 = 17

It should be pointed out that an alternative to decompression also exists for traces compressed with Sequitur. If the traces are compressed using Sequitur, they could actually be analysed without decompression. This is possible because of the nature of the Sequitur compression algorithm, which compresses the trace into a context free grammar. An algorithm for identifying hot paths of a specific length by analyzing the compressed control flow trace is given in [64]. A similar technique could also be developed for $eWPP$ to recover dependences.

4.3 Summary

In this chapter, the compression algorithms used to compress the generated traces was described along with an enhancement to the Sequitur [84] compression algorithm. Finally, detailed experimental evaluations were presented to provide evidence for the

Table 4.11. Decompression times in seconds for compressed traces

Program	Sequitur		VPC	
	WPP + cAT	eWPP	WPP + cAT	eWPP
256.bzip2	183	160	73	59
186.crafty	118	98	53	40
252.eon	201	131	49	69
254.gap	165	159	61	46
164.gzip	103	101	63	32
181.mcf	185	152	71	55
197.parser	257	243	119	79
253.perlbmk	250	193	95	75
300.twolf	242	227	112	82
255.vortex	160	117	83	50
175.vpr	195	168	83	92
Multithreaded Programs				
fmm	72	59	33	26
barnes	38	32	17	14
Average	167	142	70	55

effectiveness of extended whole program paths (eWPPs) in producing compact traces that can be efficiently stored on disk. Data indicates that the compressed eWPP can be stored in 24 % (30 %) of the space required to store the control flow and address trace when compressed using Sequitur (VPC).

Chapter 5
Generating Reduced Traces of Long-Running Multithreaded Executions

The previous two chapters described a representation to compactly store the generated data and control dependence information of a program execution on disk. This chapter describes an orthogonal technique which is to reduce the size of the generated traces. This is important because many multithreaded programs like servers are long-running and can generate very huge traces. This challenge is addressed through the *Execution Reduction* (ER) system proposed in this chapter that realizes a combination of checkpointing/logging and tracing such that traces collected contain only the execution information from those regions of threads that are relevant to the fault. Following execution reduction, the replayed execution takes lesser time to run and it generates a much smaller trace than the original execution. Thus, the cost of generating the traces and the trace sizes are greatly reduced. Notice that after the traces are generated, the compact trace representation already proposed can be used to efficiently store the traces.

5.1 Overview

Generating traces of long-running, multithreaded applications is a challenging task. Since an execution of these programs can be potentially very long, it is time and space infeasible to continually trace the execution online, i.e., when the application is running normally doing useful work. Also, off-line tracing, i.e., generating the traces by replaying the execution after it fails is also challenging for the following two reasons. First, the programs are generally non-deterministic, that is, for instance, two executions on the same input could behave differently depending on the order in which the threads were scheduled. While the bug may manifest itself in one execution, it need not do so in another execution. Second, it is still expensive to collect and store dynamic traces which may be needed to locate a bug.

To address the first problem, when designing a debugging framework for multithreaded programs, it is important to make use of a checkpointing/logging infrastructure that can precisely log the important events so that when a bug manifests itself, it can be reproduced during the replay of the current execution. Checkpointing/logging/replaying is an attractive technique, the merit of which is its capability to replay from the intermediate points of the execution at which checkpoints are created. It was invented to facilitate debugging parallel and distributed programs [85, 104]. It quickly gained popularity in general application debugging [92, 93]. A lot of research has been carried out on how to reduce its cost [99, 82] and improve its usability [96]. An integration of checkpointing/logging and tracing within a single infrastructure would be very useful. Logging will allow replaying of the fault and then traces can be collected during replay. Further, checkpoints divide the whole execution into intervals. Tracing can be applied to those intervals that corresponds to the fault. This chapter proposes a framework which can record the events of the original run and checkpoints the execution at regular intervals in order to be able to faithfully replay it from any checkpoint or the start of the program.

Although checkpointing serves to limit tracing to a portion of the execution, corresponding to a checkpoint interval, this still does not entirely solve the problem. This is because checkpointing is expensive and can be done only once every few minutes. Hence, trace sizes for a checkpoint interval can be very large. A set of multithreaded benchmark programs were collected as shown in Table 5.1 and Table 5.2 shows the sizes of control flow traces and dependence traces for sample runs of these multithreaded programs. It is clear from the data that the sizes of the traces produced is large and the runtime overhead of their collection is substantial even for a few seconds of execution. Hence, a technique is needed to address the challenge of efficiently generating traces for the execution corresponding to the relevant checkpoint intervals.

Solving the second problem by providing a fine-grained tracing mechanism that is practical for long-running programs is a far more challenging task. Support for fine-grained tracing is needed so that the fault can be later analyzed by constructing dynamic slices and the root cause of the bug can be detected. It is very important that the traces collected are small because long traces create two problems. First collection and storage of long traces is very expensive in terms of execution time and space needs. Second, the greater the amount of trace information, the greater is the effort required on part of the user to analyze the slices and locate the root cause of the bug. In the remainder of this section these techniques are discussed in more detail and a framework is proposed that can effectively address the above issues. This framework is practical due to the novel idea for generating small traces through *execution reduction* that reduces the length of the execution for replay and tracing by exploiting the observation that most threads that get executed are not directly relevant to the fault and need not be replayed or traced.

Many multithreaded and long-running applications such as server programs are event driven and, usually, a thread is spawned to service a new request from a client. Most of the threads execute independently of the other and a fault that occurs in one thread, which can even lead to a crash, is not influenced by a majority of the other threads. This observation is exploited during replay by executing only those threads that cause the fault to occur. The result is that the replayed execution is exactly what is necessary to reproduce the bug and hence, the trace that it generates is shorter and also exactly captures the program behavior that led to the fault. To find the threads that are relevant to the fault, a technique is developed that can detect the various dependences between the executing threads very efficiently in time and space. Using the dependence information between threads, the set of those threads

Table 5.1. Benchmarks and the bugs used in the Experiments.

Program	Description	LOC	Description of bugs used
mysqld	Database (ver. 3.23.56) (ver. 3.23.56) (ver. 3.23.48)	508 K	a) Mem. bug (mysql-1), reported in [3] b) Atomicity bug (mysql-2), reported in [4] c) Mem. bug (mysql-3), reported in [2]
prozilla	Download Accelerator (ver.1.3.5.1)	16 K	a) Mem. bug (prozilla-1), reported in [7] b) Mem. bug (prozilla-2), reported in [6]
proxyc	small C proxy	219	a) SIGPIPE bug (proxyc-1), Found using Change Log
axel	Download Accelerator (ver. 1.0a)	3 K	a) Mem. bug (axel-1), reported in [8]
pftp	Port File Transfer	8 K	NONE
balsa	Email client	100 K	NONE
evolution	email, address book	438 K	NONE

that contributed to the fault and those that were irrelevant to the fault is obtained. Now, tracing is done on the execution by only replaying the threads relevant to the fault.

The proposed framework essentially consists of three phases: *Logging phase*; *Execution Reduction Phase*; and *Replay Phase*. Logging Phase corresponds to the original program run during which the checkpointing and logging infrastructure is turned on. This phase produces the record of all the events, that is, the *event log*. In the case a bug is encountered, during the Replay Phase the event log can be used to replay the execution from a checkpoint or the start of the program. During the Replay Phase

TABLE 5.2. Trace sizes of multithreaded programs for small runs and their collection time in seconds.

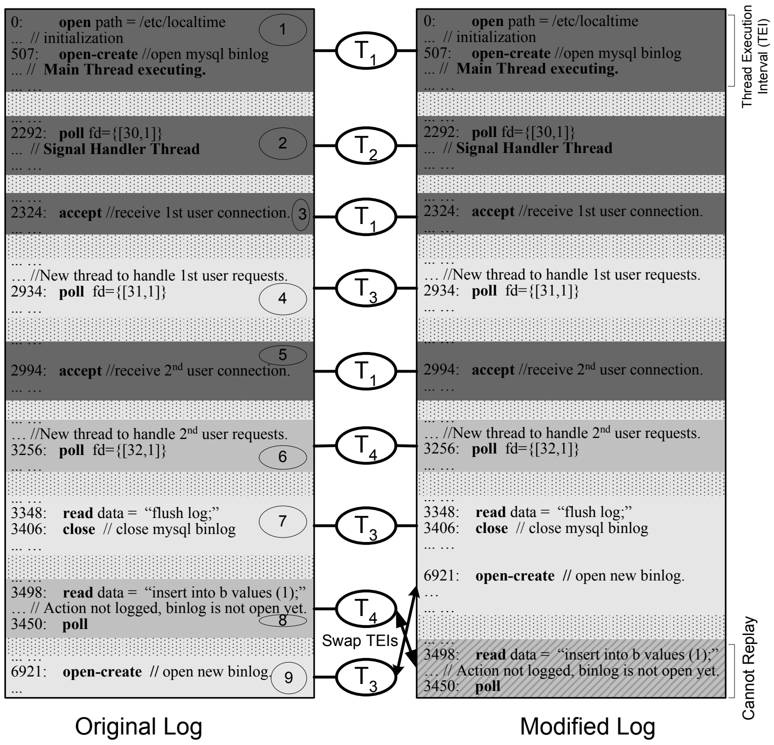
Program	Num of Threads	Exec. Time (secs.)		Control Trace	Dep. Trace
		Original	Traced		
mysql	10	13	2886	6 GB	21 GB
evolution	10	11	179	87 MB	390 MB
balsa	7	17	1787	92 MB	209 MB
pftp	3	10	903	543 MB	482 MB
proxyc	9	10	880	1360 MB	456 MB
axel	3	8	184	313 MB	456 MB
prozilla	5	8	2640	2 GB	6 GB

tracing is turned on to collect control-flow and/or dependence traces that are then used during debugging for fault location. Since the cost of the Replay Phase can be very high due to tracing, to reduce the cost of this phase it is preceded by the Execution Reduction Phase. In this phase first the dynamic dependences between the various threads are discovered. Then using this information, the subset of threads and their execution intervals that must be replayed in order to reproduce the fault are identified. Only the identified execution intervals of the subset of threads are retained for the Replay Phase and the event log is pruned to enable replay of the reduced execution. The following sections discuss the technique in greater detail.

5.2 Motivating Example

In this section, the approach is motivated using as example, a memory bug in the `mysql` database server. A known error in a `mysql` version is described and how to apply the approach of replay to trace on the faulty execution due to this bug is discussed.

According to [3], `mysql` version 3.23.56, has a memory error which is as follows. When a thread tries to load data into a table without explicitly connecting to the



Original Log

```
0:      open path = /etc/localtime
... // initialization
507:    open-create //open mysql binlog
... // Main Thread executing.
... ...
... ...
2292:  poll fd={[30,1]}
... // Signal Handler Thread
... ...
... ...
2324:  accept //receive 1st user connection.
... ...
... ...
... //New thread to handle 1st user requests.
2934:  poll  fd={[31,1]}
... ...
... ...
2994:  accept //receive 2nd user connection.
... ...
... ...
... //New thread to handle 2nd user requests.
3256:  poll  fd={[32,1]}
... ...
... ...
3348:   read data = "flush log;"
3406:   close  // close mysql binlog
... ...
... ...
3498:   read data = "insert into b values (1);"
... // Action not logged, binlog is not open yet.
3450:   poll
... ...
6921:   open-create  // open new binlog.
...
```

Modified Log

```
0:      open path = /etc/localtime
... // initialization
507:    open-create //open mysql binlog
... // Main Thread executing.
... ...
... ...
2292:  poll fd={[30,1]}
... // Signal Handler Thread
... ...
... ...
2324:  accept //receive 1st user connection.
... ...
... ...
... //New thread to handle 1st user requests.
2934:  poll  fd={[31,1]}
... ...
... ...
2994:  accept //receive 2nd user connection.
... ...
... ...
... //New thread to handle 2nd user requests.
3256:  poll  fd={[32,1]}
... ...
... ...
3348:   read data = "flush log;"
3406:   close  // close mysql binlog
... ...
6921:   open-create  // open new binlog.
...
... ...
... ...
3498:   read data = "insert into b values (1);"
... // Action not logged, binlog is not open yet.
3450:   poll
```

FIGURE 5.1. Motivation - `mysql` (Seg. Fault) Memory error, the different threads ($T_1, \ldots, T_4$) are marked and the thread execution intervals ($1, \ldots, 9$) are numbered. The reduced log shows the intervals that are replayed. The Symbol 'T' in the reduced log shows the only intervals that are traced.

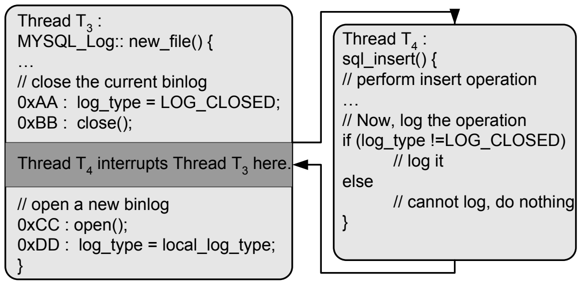
database, the field that stores the database name is accessed without checking for the *NULL* value. This causes the server to crash at this point. Consider an execution in which, after processing a set of queries, the above fault is exercised.

Logging Phase. To begin with, the server is run with light weight logging enabled, that is, the events are logged and checkpoints are performed at fixed intervals. Figure 5.1 shows the events recorded in the event log. T_1, T_2 T_3, and T_4 refer to four unique threads created. The event log also shows the points where a thread is descheduled and another thread is scheduled. A region in the log corresponding to the maximal set of consecutive events from the same thread is referred to as a *thread execution interval* (TEI). The log in Figure 5.1 shows 9 thread execution intervals.

The queries and the activities of the corresponding threads are as follows:

- Thread T_1 is the startup thread that handles new connections and creates threads to service requests.
- Thread T_2 is created by T_1 to handle signals.
- Thread T_3 is created by T_1 to handle a user. This user first looks at all the databases and then crashes the server by issuing the "load" command.
- Thread T_4 is created by T_1 to handle another user. This user does a "select" operation on table 'b' in database 'test'.

The server crashes at TEI 9 and the program is taken to the next phase of the framework. Figure 5.2 shows the root cause for this bug. Notice how the database field(thd→db) is accessed without checking for an invalid value. The fix is to place a check before the access and report an error if the value is invalid, instead of crashing.



```
Thread T3 :
MYSQL_Log:: new_file() {
…
// close the current binlog
0xAA :  log_type = LOG_CLOSED;
0xBB :  close();
Thread T4 interrupts Thread T3 here.
// open a new binlog
0xCC : open();
0xDD :  log_type = local_log_type;
}
```

```
Thread T4 :
sql_insert() {
// perform insert operation
…
// Now, log the operation
if (log_type !=LOG_CLOSED)
        // log it
else
        // cannot log, do nothing
}
```

FIGURE 5.2. Source Code - MySql Memory Error, root cause

Execution Reduction Phase. Once it is known that a fault has occurred (e.g., program crash has occurred), the next step is to replay the fault and collect the trace during replay to assist the user in debugging. However, first the execution reduction phase is used to determine the subset of computation that needs to be replayed and traced.

The first step in this phase is to identify and remove the threads from the event log that did not contribute to the bug. This reduces the execution time of the program and keeps the resulting traces small. Alternatively, the trace was generated for the entire execution by replaying it with the tracing turned on, then the resulting trace sizes are as high as 16 GB for 15 seconds of execution. It is easy to see that if the programs had run for a long time before the fault occurred then the trace sizes would become unmanageable.

Consider the different threads in the example shown. Notice that thread T_4 is irrelevant to the fault. Intuitively, it can be seen that the bug would have still occurred even if the second user did not exist. The queries corresponding to the execution of thread T_4 are completely independent of the queries corresponding to T_3. Hence, all the events from the replay log corresponding to T_4 is removed. In the reduced log in Figure 5.1, notice that the TEIs 5, 6, and 8 do not exist. These correspond to the creation and execution of thread T_4 in the original log.

Now that irrelevant threads have been removed, the next step in this phase is to identify irrelevant TEIs. All the remaining threads are relevant to the bug but not all of their TEIs are relevant. In Figure 5.1, TEI 7 corresponding to T_3, which is the execution corresponding to a "show databases" query from the user, is irrelevant to the fault. Hence, this is also removed from the replay log. The reduced log now shows only the relevant TEIs that caused the fault. Replaying the program using the reduced log generates the fault much faster.

Replay Phase. In this final phase the program is replayed using the reduced log with the tracing infrastructure turned on. Since the execution contains only the necessary TEIs, the traces produced are much smaller. The size of the traces are further reduced in this phase by exploiting the following observation. *Even though all the TEIs in the reduced log have to be replayed to produce the fault, not all of them have to be traced.*

For instance, thread T_1 is present only to create the faulty T_3, and thread T_2 to handle signals. The code that is executed in T_1 and T_2 does not contain the root cause. Hence, only thread T_3 needs to be traced, that is, TEIs 4 and 9 in the reduced log. The original trace is reduced in size by 99.99% and this reduced trace captures the root cause as desired. The user then inspects the generated trace and discovers the root cause of the bug.

5.3 Automated Execution Reduction

In this section the types of dynamic dependences that must be found to automatically perform execution reduction is identified. Also, efficient dynamic algorithms used to identify these dynamic dependences are proposed. Before presenting the above details, the types of events that are recorded in the replay log are discussed.

When the application is being executed in the real world environment, it is executed with a lightweight checkpointing/logging mechanism turned on. Non-deterministic events are recorded as and when they happen. In addition, the program is checkpointed at regular intervals. The logging is to ensure that the program can be replayed to reproduce a fault if one occurs. The following are some of the events that must be recorded in the log in order to correctly replay the execution of the multithreaded program. The *thread scheduling* events of a multithreaded program are the most important events that need to be captured since they can vary from one execution to another. The replay log captures all the events where a thread was descheduled and a different thread was scheduled. It should be noted that the order in which the different threads access shared memory, which must be preserved to successfully replay a multithreaded program, need not be captured explicitly. By recording the scheduling information, it is ensured that when the program is replayed as per the schedule, the order of shared memory accesses by the different threads does not change. This holds only when the user level threads are executing in a uniprocessor environment. Notice that the order in which a single thread accesses memory is preserved in the control flow of the execution and hence, need not be explicitly recorded. To summarize, by preserving the thread scheduling of the original execution it can be guaranteed that the order of memory accesses in the replayed program is exactly the same as the original program. Some of the other important events that need to be captured are external events like signals, interrupts, and IO reads and writes. Reads and writes to files need to be logged along with the file offsets and the size of the read or write in order to undo these operations and restore the original file contents when commencing replay.

When a fault is encountered, execution reduction is carried out before tracing the execution. Execution reduction is critical because even if the program is replayed from the most recent checkpoint, the trace that is generated can be very long and its generation can take a long time. Execution reduction is based upon two types of information: identification of irrelevant threads; and identification of irrelevant thread execution intervals. The algorithms for identifying irrelevant threads and irrelevant thread execution intervals are described in the following subsections.

5.3.1 Discovering Irrelevant Threads

Lets consider the problem of identifying threads that are irrelevant to the fault, that is, the execution of these threads does not influence the execution of the threads that resulted in the fault. To achieve this goal, the interactions or dependences between the different threads have to be known. Thread interactions can take place via *events*, *files*, or *shared memory*. Examples of event interactions are actions such as thread creation and join involving two or more threads. File interactions occur when threads communicate by reading and writing from files. The most frequent type of thread interactions are through the use of shared memory regions.

To detect whether a thread is relevant or irrelevant to the fault, the information about the three kinds of dependences between the threads needs to be obtained. To replay a thread T_i, another thread T_j has to be replayed if and only if thread

T_i depends on thread T_j. Once the dependence information is obtained, a *Thread Dependence Graph* (TDG) can be constructed. Then, the set of relevant threads are identified, $REL(T_i)$, for any thread T_i by traversing the TDG. To replay the execution of thread T_i exactly, it is necessary and sufficient to only replay those threads that are in the set $REL(T_i)$. Given this information, all the threads that are irrelevant to the faulty thread can be identified and eliminated by pruning the replay log. Then only those threads that contributed to the fault remain, resulting in reduction in the execution. The definitions of TDG and Relevant Threads are given below.

Definition 1.(*Thread Dependence Graph (TDG)*) The Thread Dependence Graph of a multithreaded program execution, $TDG(N, E)$, consists of a set of nodes N and a set of directed edges E where each node $n_i \in N$ corresponds to a unique thread T_i that was created in the current run and each edge $(m_i \rightarrow n_j) \in E$ indicates that there is a dependence path from thread T_i to thread T_j, that is, T_j is dependent on thread T_i. Also, each edge is annotated with one or more of the symbols in the set $\{File, Event, SharedMem\}$ to indicate the type of dependence(s).

Definition 2.(Relevant Threads) The set of relevant threads corresponding to a thread T_i, $REL(T_i)$, is defined as $REL(T_i) = \{T_j | T_j \in \Gamma$ and $\exists$ a dependence path from T_j to $T_i\}$ where Γ is the set of all threads in the current run.

Next, each of the three types of interthread dependences are discussed in detail. In particular, how each of these dependences is identified is explained.

Event Dependences. The replay log contains explicit information on all the thread events (e.g., thread creation and termination, synchronization events such as join, etc.) and hence the log can be analyzed to obtain all event dependences between threads. Hence, by inspecting the log and looking at the records corresponding to these events, the various event dependences between the different threads can be detected. For example, in the replay log in Figure 5.1, the events in TEI 1 corresponding to thread T_1 indicate that a new thread T_2 was created and scheduled to run in TEI

2. Hence, it can be inferred that thread T_2 is dependent on thread T_1 by the parent-child relationship. Notice that this means T_1 has to be replayed in order to replay T_2 whereas the reverse is not true. *In summary, the replay log captures all the event dependences between threads and a simple scan of this log is enough to discover all of them.*

File Dependences. Now, discovering dependences between threads due to file operations is discussed. A file data dependence exists from thread T_i to thread T_j if thread T_j reads from an offset in any file F that was written to by thread T_i. This implies that for successful execution of thread T_j, thread T_i must also be replayed. Dependences between threads due to files can be directly obtained from the replay log. The replay log records information on the files that were read or written by every thread, the offsets from which the reads and writes took place and the size of the operation. This is done primarily to restore the contents of the files while commencing replay. *Hence, by scanning the replay log all file dependences between threads can be retrieved.*

Shared Memory Dependences. Now, discovering the most common interactions between threads that result in shared memory dependences are discussed. There exists a shared memory dependence from thread T_i to thread T_j if T_j reads a value from any memory address 'a' that was written by T_i. Notice that this is a RAW dependence. Also notice that this is the only dependence that needs to be captured if the program is instrumented to convert all WAW and WAR into RAW. Thread T_j is dependent on T_i since T_i either generates the value and has to be replayed for successfully replaying T_j (traditional RAW) or T_j and T_i have a potential data race in the form of WAW or WAR.

Shared Memory dependences between threads cannot be simply obtained from the replay log. Recall that to make the logging scheme lightweight, explicit capturing of information unnecessary for replay must be avoided. By capturing the thread schedule in the log, the need of capturing the shared memory dependences does not arise. Therefore, to obtain shared memory dependences, the program must be replayed and track these dependences as they occur using a mechanism for detecting shared memory dependences. This mechanism must track shared memory dependences between threads and output thread ordered pairs (T_i, T_j) that are involved in at least one shared memory dependence. Note that to construct the TDG, every occurrence of a dependence between a pair of threads need not be output.

It is desirable that the technique that detects shared memory dependences has low overhead. Even though this phase is carried out in the debugging stage of the program, unreasonable delays is not desirable to the user who is debugging the code. One approach that does not involve runtime overhead could be based upon static analysis [97, 94]. This approach has the disadvantage of producing a conservative TDG using which fewer threads may be identified as being irrelevant. Another issue is that dynamic opportunities for eliminating irrelevant threads will be lost. During a given execution, a potentially shared memory region may however, be accessed by just one thread. The static approach cannot take advantage of such opportunities. However, a well designed dynamic approach can take advantage of this information to identify more irrelevant threads.

Let us first consider a *naive* dynamic strategy for detecting shared memory dependences. A *hash table* can be used to maintain, for each address 'a', the *thread id* of the thread that performed the *most recent write* to 'a'. To detect an interthread dependency, when a load operation is performed on address 'a' by thread T_j, the thread id T_i that wrote to it last is obtained from the *hash table* and an interthread dependence is formed if T_i and T_j are different. Although this scheme is straightforward, it is inefficient in both space and time. It is space inefficient because the size of the hash table is as large as the memory footprint of the original program. For large applications, this could potentially run out of memory. It is time inefficient because every load and store that executes must access the hash table. Every store must write the thread id to the corresponding hash entry and every load must read it from the hash table. Next a scheme is presented that greatly improves the efficiency of the above naive interthread dependence detection scheme. This scheme is efficient in both space and time. This scheme is based upon two optimizations that achieve elimination of majority of the expensive hash table lookups.

The first optimization introduces a new look-up table, called *RegionMap*, such that accesses to this new table are less expensive than accesses to the *hash table*. Often times, the dependence is resolved by accessing the *RegionMap* and hence the need for accessing the *hash table* is eliminated resulting in savings in time. In addition, we will see that the size of the hash table is greatly reduced.

Let us discuss the *RegionMap* in greater detail. The 32-bit virtual memory address is divided into two parts: the higher order 16-bits act as a *region specifier*; and the lower order 16-bits are used as the *offset* address within the region. The region itself can be either a *shared memory region* or a *non-shared memory region*. The *RegionMap* is indexed by the 16-bit region specifier and it contains a bit for every region, called *isSharedMem*, that indicates if the region has been *dynamically observed* to behave as a shared memory region or not. All region bits are initially set to *False* implying that all regions are non-shared memory to start with. The region bit for a region is set to *True* if more than one thread accesses the region. The region table also contains a field, called *firstThread* that stores the identifier of the first thread that wrote to it. This is initially set to an invalid value and is initialized by the *first thread* that writes to it.

Now, it is shown how an access to the hash table may be avoided by first accessing the *RegionMap*. For a load operation, a look up is done on the *RegionMap* first to see if it is an access to a shared memory region or not. If the region is currently indicated to be non-shared memory, nothing needs to be done any further as this load does not involve an interthread dependence. However, if it is a shared memory

```
do_for_every_load_and_store(ThreadId currThread, Address a){
/* RegionMap, array of $2^{16}$ entries, each entry has 2 fields
   isSharedMem bit, firstThread field - initialized to 0 */
// prevThread,prevRegion,prevSharedMem - prev. load / store
currRegion=$a$ >> 16; // higher order 16 bits
Stage I :
   if(prevThread=currThread && prevRegion=currRegion
      && prevSharedMem=False)
      return;
   prevThread=currThread; prevRegion=currRegion;
Stage II :
   entry = RegionMap[currRegion];// Lookup Region table
   if(entry→isSharedMem=False)
      if(entry→firstThread=0)
         prevSharedMem=False;
         entry→firstThread=currThread; return;
      if(currThread = entry→firstThread)
         prevSharedMem=False; return;
      else
         prevSharedMem=True;
         /* stores update shared memory bit
         loads check for dependence with first thread */
         if(load instruction)
            return;
Stage III :
   if(store instruction)
      write_threadid_into_hash_entry(a);
   else //load instruction
      threadId = read_threadid_from_hash_entry(a);
      if(threadId is valid)
         Track_Dependence(currThread, threadId);
      else
         Track_Dependence(currThread, entry→firstThread);
   return;
}
```

FIGURE 5.3. Pseudo-code for detecting shared memory dependences. The code shows the processing that is done for every memory load and store instruction. The 3 stages are clearly marked.

region the *threadId* of the store operation involved in the dependence is obtained by looking up the *hash table* and checked to see if it is an interthread dependence. For a store operation, if the region is shared memory, an update is performed to the hash entry corresponding to the memory address with *threadId*. If the region is not shared memory, a check is done to see if the thread performing this store could potentially make it a shared memory region, that is, the current thread's id is compared with *firstThread* id to see if they are different. If they are different, a shared memory region has been detected and the region bit, *isSharedMem*, for this region is set to *True* in the *RegionMap*. The hash entry for the 32-bit address is also updated. However, if the region is still not shared then nothing further needs to be done.

From the above operation of the *RegionMap*, and the *hash table*, the following has been achieved. *The size of the hash memory now at most equals the combined sizes of only the shared memory regions and not the total virtual space used.* Hence, this is a huge saving and for the many programs that were looked at, the amount of shared memory that is used is much less than the actual memory used. Also, *for loads and stores that do not access shared memory regions, the expensive hash table lookup operation is avoided.*

The second optimization is designed to further reduce the runtime overhead by reducing the *RegionMap* lookups and replacing them with cheaper operations. In this sense this optimization is analogous to the first optimization which reduced the runtime overhead by replacing some of the expensive hash table lookups by cheaper *RegionMap* lookups. This second optimization exploits the *locality in the regions accessed* by most loads and stores. In particular, locality here refers to the characteristic that consecutive executions of the same static load (store) often involve the same region. When this is the case, handling of one region access by a load (store) makes the handling of the next access to the same region by the same load (store) redundant.

Finally, a three stage algorithm is proposed for handling each load and store such that first stage is the cheapest and the last stage (hash table lookup) is the most expensive. While in general a load or store may have to go through all three stages, very often this is not the case and hence the runtime overhead of the three stage scheme is greatly reduced when compared to the runtime overhead of a single stage scheme involving hash table lookup. Next all of the ideas are put together into a three stage algorithm described below (pseudocode is given in Figure 5.3).

Stage I - Check region of previous memory operation. In this stage, for the load or store that is being processed, if the previous memory operation (load or store) was from the same thread, it accessed the same region, and was found to be non-shared, then it can be guaranteed that this region will continue to remain non-shared. (The variables *prevThread*, *prevRegion* and *prevSharedMem* contain this information about the most recent load/store operation.) Hence, a *RegionMap* lookup is not required and no further processing of this memory operation is required. Due to significant locality of regions, over half of the loads/stores did not proceed beyond this stage.

Stage II - Check RegionMap table and update isSharedMem bit. In this stage, and access to the *RegionMap* is required as the locality check in Stage I failed. The *RegionMap* tells us if the region accessed is shared memory or not. For a load or store, if this region is not shared memory then a operation on the hash table is unnecessary. However, a check needs to be done to see if this thread's access could potentially make it shared memory and flip the *isSharedMem* bit accordingly.

Stage III - Access the hash table. In this stage, it has been determined that the region is shared memory by looking up the region table and therefore the expensive hash accesses are performed. For a load operation the hash table is accessed to retrieve the thread that wrote to this 32-bit address last and check if it is an interthread dependence. The function *Track_Dependence* does this check. However, if this address was last written to by a thread when this region was not detected to be shared memory, then the contents of the hash memory would be invalid as the thread that wrote to it last did not create the hash entry. In this case, the *firstThread* field of this region is accessed and the dependence is now obtained. For a store operation, the hash entry corresponding to the 32-bit address is updated.

TABLE 5.3. Cost of shared memory dependence tracking for some multithreaded programs.

Program	Staged Tracking		Time	Memory Used	
	Stage I %Ld+St	Stage II %Ld+St	Staged/ Naive	Naive	Staged
mysql	52 %	10 %	55 %	3.6 MB	0.8 MB
evolution	16 %	10 %	64 %	8.4 MB	5.9 MB
balsa	67 %	15 %	73 %	8.1 MB	1.8 MB
proftp	50 %	0 %	50 %	3.3 MB	3.3 MB
proxyC	72 %	16 %	56 %	6.6 MB	1.3 MB
axel	50 %	18 %	12 %	0.3 MB	0.1 MB
prozilla	56 %	19 %	41 %	1.2 MB	0.3 MB
Average	52 %	13 %	58 %	4.5 MB	1.9 MB

To evaluate the dependence tracking technique, experiments were conducted on some multithreaded long-running programs and measured the percentage of loads and stores that terminated at each stage. Table 5.3 shows the data. It also shows the space overhead of this approach. From this data it can be seen that on average 52% of all loads and stores terminate at Stage I, that is, they do not require a *RegionMap* or a *hash table* access. Additional 13% terminate in Stage II. Thus, finally, on an average, only 35% of all loads and stores performed hash accesses as they reached Stage III. The runtime overhead of the staged approach is 58% of the naive tracking scheme. Also, on an average, the total memory used by the Staged Tracking approach is only 42% of the memory used by the naive approach.

Note that the region size, which is 16 bits now, can be varied to be coarser or finer.

By making it finer, shared memory space could be determined much more accurately but the locality optimization would not be as effective. Notice that a region size of 32 bits is basically equivalent to the naive approach. Making the region size coarser could give more opportunities for locality but more regions would become shared memory and hence the locality benefits might not be useful. Hence, the region size is a trade-off between how fine shared memory can be detected and how much locality can be obtained. A 16 bit sized region was found to work well with the benchmarks.

Eliminating Irrelevant threads. At this point the complete thread dependence graph is present with all dependences detected and annotated. The set of threads that are relevant is detected to correctly replay the fault. The rest of the threads are irrelevant. The replay log is pruned to remove all the records corresponding to the irrelevant threads. Now, the reduced replay log has only information on relevant threads and the execution has already been shortened.

5.3.2 Discovering Dependences Across TEIs

Now that the threads that are irrelevant to the fault have been eliminated, the next step is to eliminate irrelevant thread execution intervals (TEIs) from the relevant threads. For this step, the interactions between the various TEIs have to be detected. Notice that the information on the dependences between TEIs that correspond to different threads is already present. Now, the event, file, and memory dependences between TEIs belonging to the same thread have to be found. Event and file dependences across TEIs of the same thread are found using the original replay log. To find memory dependences, the program is replayed again, but using the reduced replay log, and the naive approach that is described in the last section is used as the execution has been shortened already.

Now, just as irrelevant threads were detected by using the TDG, analogously, a dependence graph for TEIs is constructed to remove all irrelevant TEIs. The reduced replay log is further pruned to remove all records corresponding to the irrelevant TEIs. A highly reduced log that contains only relevant TEIs is now obtained. This completes the second phase of the framework.

5.3.3 Selective Tracing of Reduced Execution

The reduced replay log contains only those thread execution intervals that need to be replayed. *However, not all TEIs have to be traced.* For instance, a thread's execution trace, that merely created the faulty thread which has a memory error, is not useful as the invalid memory access could not have come from this thread. All such TEIs are identified using the event, file, and shared memory dependence information that is available between the various threads and TEIs. During replay, tracing is turned on when a TEI needs to be traced and turned off otherwise. The overhead of toggling tracing is low as it is done at the granularity of TEIs. At the end of this stage, a trace of the faulty execution is obtained that is short and contains the root cause of the bug.

5.4 The Execution Reduction System

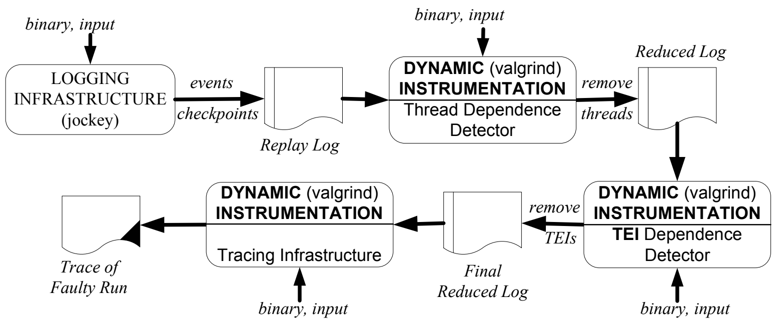


FIGURE 5.4. Implementation of the system showing each step of the framework.

In this section, the implementation of the ER system is described that incorporates checkpointing/logging, dependency detection, and (selective) tracing. This system was used to analyze several bugs in long-running multithreaded programs.

Figure 5.4 shows the system. The system consists of a logging component whose main role is to log the events of the original execution and also create checkpoints at regular intervals. The system's key component is the dynamic instrumentation engine. It is involved in many steps of the debugging process. It uses the information in the replay log created by the logging infrastructure to replay the multithreaded program exactly. Also, while replaying the program, it can dynamically instrument the binary to detect dependences and collect traces. The information it generates is used to shorten the replay logs by pruning irrelevant threads and TEIs. Since the instrumentation is dynamic, tracing can be turned on and off at run-time. The following paragraphs discuss the tools used to perform logging and dynamic instrumentation.

Logging/Checkpointing Infrastructure. The *jockey* user level library [96] has been used to perform checkpointing and logging for replay. *jockey* is a very powerful system that works on most multithreaded programs and is also very easy to use. During execution, even before the application can execute, *jockey* takes control and scans the application binary for system call instructions. It then redirects these calls to a *jockey* handler and lets the application execute. During system calls, *jockey* logs events, scheduling decisions, creates checkpoints, etc. Scheduling of the user-level threads can be controlled by *jockey* because it uses its own thread libraries and any current thread is descheduled only at a system call boundary. Checkpointing is achieved by retrieving the layout of the application's virtual space and dumping all virtual memory segments that belong to the application. To summarize, *jockey* related work is performed only during a system call and otherwise, the application executes as though it was unaware of *jockey*. Since *jockey* works only for uni-processor systems, it cannot log the execution of multithreaded programs that run on multiprocessors. However, the approach described is general and by using a logging mechanism for multiprocessors [78, 109], the execution reduction techniques can be applied to programs that execute on multiprocessors.

Dynamic Instrumentation Engine. To perform dynamic instrumentation, the *valgrind* [80] system has been used which can handle x86 binaries. The binary is executed with *valgrind* which calls an instrumentation function just before a basic block is to be executed for the first time. The instrumentation transforms the basic block and rewrites the code cache with the instrumented basic block so that future calls to execute this basic block does not have to go through the instrumentation process. The code cache of a basic block can be invalidated which will cause the instrumentation function to be called when this basic block executes again. Here, the instrumentation could either be modified or turned off . Hence, the instrumented code can be dynamically manipulated.

The dynamic instrumentation engine forms the core of the framework. Its first job is to parse the log generated by *jockey* and replay the program. For multithreaded programs, the scheduler decisions are the most important events that have to be replayed. The schedules are replayed as follows. The scheduling decisions in the original program are made only at system call sites by *jockey*'s thread library. When logging the schedule, the system also logs the number of system calls that the thread executed since it was scheduled and before it was descheduled. In *valgrind*, for a thread that is currently executing, the system uses the number of system calls executed to decide when to deschedule the thread. Upon reaching that system call (*valgrind* has event handlers that are called before and after a system call which is used to count system calls), the system forces the scheduler to deschedule this thread and switch to the appropriate thread. Hence, it can be guaranteed that the threads will be scheduled according to the replay log. As mentioned before, preserving the schedules will also guarantee that the shared memory dependences of the original execution will be preserved. File events can be replayed exactly if the contents of the modified files are restored. There are some system calls for which *jockey* saves the contents of the original run. For example, in a server program, if a client makes a connection request, the contents of the *socket-read* system call are saved in the *jockey* log. During replay, when the system call *socket-read* is to be executed, *jockey* will return the saved contents instead of executing the system call. What is done is exactly the same in *valgrind* during replay of the program. When the *socket-read* system call is reached, the system does not perform the system call but returns the contents saved in the *jockey* log. For the programs considered, the events that were handled were enough to replay the execution.

Given that the program can be successfully replayed using *valgrind*, it is now used to detect shared memory dependences, TEI dependences and, finally, obtain traces. In the thread execution reduction phase, the system replays the program and instrument the loads and stores in every basic block according to the algorithm described in Figure 5.3. Once it has obtained the shared memory dependences, it can now find the irrelevant threads of this faulty execution. Note that the file and event dependences are already available in the replay log. The system uses this information to prune the replay log. Similarly, it finds dependences across TEIs and further prunes the log. Once the final reduced log is obtained, *valgrind* is used to trace the shortened execution and output the trace. Here, since it does not trace all replayed TEIs, *valgrind*'s ability is used to selectively switch on or off the tracing for a particular TEI.

5.5 Experiments

The multithreaded benchmark programs used in the experiments were already described in Table 5.1. For all these programs, the trace sizes and the cost of detecting shared memory dependences have been already shown in Tables 5.2 and 5.3 respectively. Now, for studying the effectiveness of the entire ER system, experiments were performed with only the buggy versions of these programs.

Table 5.4. Trace sizes (Basic Blocks) produced by the original and shortened runs (M - million, B - billion).

Program	Number of Basic Blocks				
	Orig.	R_Thread	R_TEI	SR_TEI	Orig./SR_TEI
mysql-1	976 M	19349	16695	1964	490000
mysql-2	733 M	1.1 M	29809	29809	24500
mysql-3	857 M	122 M	24834	9511	90100
prozilla-1	536 M	106749	81179	81179	6600
prozilla-2	764 M	764 M	764 M	1.6 M	478
proxyc-1	200 M	23736	23736	23736	8400
axel-1	55.4 M	7734	7734	1622	34000

For each bug, the following execution scenario was created. The buggy version of the program was taken and a reasonably long-running execution is created at the end of which the bug triggers the failure. For example, in *mysql*, a number of clients were invoked which were made to issue queries to the different databases created. Some of the query operations used were among the common ones like *select*, *join*, *insert*, *delete*, *orderby*, etc. At the end a client was made to perform the query operations that causes the bug to occur. The length of the execution has been limited to be around 10 seconds for *mysql* and *proxyc*. For prozilla, the length of the execution is around 5 to 7 seconds. For *axel*, the bug that was considered happens during the initialization phase. Hence, this program could not be made to run as long as other programs. Note that even though checkpointing is supported in the system, given the lengths of executions, checkpoints were not created. The following subsections discuss the different experiments that have been conducted.

5.5.1 Space Overhead

Tables 5.4 and 5.5 shows the size of the basic block (control flow) and dependence traces in terms of the number of basic blocks and dependences for the various execu-

Table 5.5. Trace sizes (Data Dependences) produced by the original and shortened runs (M - million, B - billion).

Program	Number of Data Dependences				
	Orig.	R_Thread	R_TEI	SR_TEI	Orig./SR_TEI
mysql-1	1.5 B	30375	25391	3175	470000
mysql-2	1.1 B	1.27 M	49263	49263	22000
mysql-3	1.3 B	188 M	40869	17929	73000
prozilla-1	720 M	135466	123918	123918	5800
prozilla-2	1 B	1 B	1 B	2.6 M	380
proxyc-1	56 M	6513	6513	6513	8600
axel-1	53.9 M	5119	5119	1156	46600

tions considered. What was measured was the basic block and dependence trace sizes for four different executions of the same program. The WAW and WAR dependences for these programs were a very small percentage ($< 0.01\%$) of the total number of dependences. First, the trace sizes of the original run was measured shown under the heading *Orig* in Tables 5.4 and 5.5. Then, the trace sizes of the programs by replaying only the *relevant threads* was measured which is shown under the heading *R_Thread*. The data under the heading R_TEI corresponds to the trace sizes by replaying only the *relevant thread execution intervals* in the program. For R_Thread (R_TEI), the basic block traces are smaller than the original by factors ranging from 1 (1) to 50442 (58460) and the dependence traces are smaller by factors ranging from 1 (1) to 49300 (59000).

An additional experiment was also performed which was measuring the trace sizes by using *selective tracing* (SR_TEI), that is, all the relevant TEIs were replayed but do not necessarily trace all of them. Selective tracing of TEIs is performed as follows. For programs with a memory bug, that causes a Segmentation Fault, the bug manifests itself from the root cause to the crash point through a series of memory dependences in the program. Hence, if the faulty interval TEI_i is not memory dependent on another interval TEI_j, then the trace of TEI_j does not contain any useful information about the crash. However, TEI_j may still have to be replayed since TEI_i may be dependent on it due to event dependences. Since there is already information on all the dependences between the various TEIs, this is used to decide which TEIs to trace. Then, the dynamic tracing infrastructure is used to selectively turn on tracing for the appropriate TEIs. With selective tracing, the reduced basic block traces are smaller than the original by a factor of 478 to 490000. The corresponding reduction factors for dependence traces range from 380 to 470000. Note that this huge reduction in trace sizes comes from both execution reduction and selective tracing. For *prozilla-2*, selective tracing is the only single contributing factor.

5.5.2 Time Overhead

Table 5.6 gives the data on the runtime performance for the various executions considered. First, the *logging overhead* on the original execution was measured. Table 5.6, under the heading of Logging Overhead, gives the execution times of the program run without logging (*Orig*-1) and with logging (*Logged*). The ratio of the two in column *Logged*/*Orig*-1 shows that the program execution slows down by a factor ranging from 1.1 to 2.8. The logging overhead is small for *mysql* and *axel* and slightly higher for *prozilla*, *proxyc*, and *proftp*. The reason for the slightly increased overhead for some programs is because their long-running execution involves downloading large files from a website. Jockey makes a separate copy of the contents of the downloaded file and this increases the overhead. However, the overhead is still reasonable and is acceptable to have logging turned on during normal execution.

Then,the execution times of the programs during replay from corresponding logs both without and with tracing was measured. In each of these two cases three measurements were made: the execution time to replay the *entire execution* (*Orig*-2/3); the execution time to replay the execution of only the *relevant threads* (*R_Thread*-2/3); and the execution time to replay the execution of only the *relevant thread*

TABLE 5.6. Overhead of logging and the running time in seconds of the original execution and the reduced execution with and without tracing.

Program	Logging Overhead			Replay without Tracing		
	Orig-1	Logged	Logged/ Orig-1	Orig-2	R_Thread-2	R_TEI-2
mysql-1	14.8	16.8	1.1	16.4	0.1	0.08
mysql-2	12.3	14.0	1.1	12.6	1.1	0.1
mysql-3	13.9	15.8	1.1	15.4	2.3	0.09
prozilla-1	4.8	13.4	2.8	12.4	0.08	0.05
prozilla-2	7.2	18.7	2.6	16.5	16.5	16.5
proxyc-1	11.0	19.8	1.8	16.6	0.07	0.07
axel-1	0.15	0.16	1.1	0.14	0.02	0.02

Program	Replay with Tracing				
	Orig-3	Orig-3/ Orig-2	R_Thread-3	R_TEI-3	SR_TEI-3
mysql-1	3736	227.8	0.8	0.7	0.67
mysql-2	2806	222.6	4.0	0.9	0.9
mysql-3	3270	212.3	468	0.9	0.9
prozilla-1	2664	214.8	0.6	0.5	0.5
prozilla-2	2364	143.3	2364	2364	560
proxyc-1	960	57.8	0.3	0.3	0.3
axel-1	3.2	22.8	0.3	0.3	0.26

execution intervals (R_TEI-2/3). In case of replay with tracing, an additional measurement was made that takes advantage of selective tracing (SR_TEI-3).

Consider the performance of replaying the original and reduced executions *without tracing* turned on. Excluding *prozilla-2*, while the original execution time $Orig$-2 that includes all threads ranges from 0.14 to 16.6 seconds, the execution time R_Thread-2 which excludes irrelevant threads ranges from 0.02 to only 2.3 seconds. Then, if irrelevant TEIs are removed, the execution time is further reduced to R_TEI-2 which ranges from 0.02 to only 0.1 seconds. With the exception of *prozilla-2*, all the buggy programs have a significant reduction in their execution times.

Table 5.7. Replay Log Sizes of original and shortened runs, (M - million).

Program	Number of events in replay log			
	Orig.	R_Thread	R_TEI	Orig./ R_TEI
mysql-1	4801	281	236	20.3
mysql-2	3749	489	365	10.3
mysql-3	5453	902	332	16.4
prozilla-1	7.2 M	621	73	98000
prozilla-2	10.8 M	10.8 M	10.8 M	1
proxyc-1	32.8 M	798	798	41000
axel-1	1695	954	954	1.8

Next consider the performance of the various executions *with tracing* turned on. The overhead of tracing given by column $Orig$-3/$Orig$-2 is as high as 228 which is the factor by which the execution slows down. For *mysql* the data shows that tracing can cause a significant slowdown in performance which cannot be tolerated even during debugging. However, after execution reduction this overhead is greatly reduced. Excluding *prozilla-2*, while the original execution time $Orig$-3 that includes all threads ranges from 3.2 to 3736 seconds, the execution time R_Thread-3 which excludes irrelevant threads ranges from 0.3 to 468 seconds. Then, if irrelevant TEIs are removed, the execution time R_TEI-3 is further reduced and it ranges from 0.3 to only 0.9 seconds (excluding *prozilla-2*). With selective tracing the execution time SR_TEI-3 for *prozilla-2* is greatly reduced, that is, from 2364 to 560 seconds. Thus, the combination of removing irrelevant threads, removing irrelevant TEIs, and performing selective tracing proves effective for all programs.

Table 5.7 gives the number of events in the original and reduced replay logs for the original and shortened executions. The final reduced log is smaller than the original by factors ranging from 1 to 41000 which translates into smaller execution times as already observed and hence smaller trace sizes.

5.6 Summary

In this chapter, the *execution reduction system* was described that can effectively combine checkpointing and tracing in order to debug long-running multithreaded programs. The proposed system uses dynamic techniques for eliminating the execution of irrelevant threads and irrelevant thread execution intervals from the final replay phase that collects traces. Further, it also eliminates unnecessary tracing during the replaying of relevant threads and thread execution intervals. The combined effect of the above approach is that the tracing overhead and the amount of trace data collected is greatly reduced. Most importantly, to make the above scheme work, a three stage scheme was developed for identifying dynamic shared memory dependences between executing threads that is both space and time efficient. Detailed experiments demonstrate the effectiveness of the proposed techniques and the data shows that the sizes of the generated traces can be reduced by two to five orders of magnitude and the tracing time by two orders of magnitude.

Chapter 6
Environmental Fault Avoidance Via Execution Perturbation

As mentioned earlier, for critical applications, bringing down the application and waiting till the fault is fixed might not be acceptable in some situations. Hence, in such cases techniques for fault avoidance is attractive which let the application continue execution inspite of the fault. This chapter discusses a technique to avoid a class of faults in programs. There are certain errors in a program that manifest as a fault only under certain environment conditions. For instance, a bug in a thread synchronization part of a multithreaded program may be exposed only under certain thread scheduling events. Such class of faults are referred to as *environment faults* as they occur only when certain conditions prevail in an execution environment and can be avoided by appropriately modifying the environmental conditions. This chapter proposes a framework to capture and recover from *environment faults* when they occur and to prevent them from occurring again. Three different types of environment faults have been investigated that can be avoided by altering the execution environment (atomicity violation, heap buffer overflow, and malformed user request) and the framework has been found to be effective in avoiding them.

6.1 Overview

A large number of faults that occur in today's software are due to the execution environment. In a study by Chandra and Chen [30] and mentioned in Qin et al. [89], 56% of faults in the Apache server are dependent on the environment. The faults that can be averted by modifying the execution environment is referred to as *environment* *faults*. These faults can be non-deterministic. For instance, synchronization faults in multithreaded programs are non-deterministic as they may not occur for some thread schedules and can be averted by avoiding the thread schedules (the environment) that expose the fault. They could also be deterministic as in the case of some heap buffer overflow faults which can be avoided if the memory allocator (the environment) sufficiently pads the allocated heap memory.

This chapter presents an online framework to capture and recover from environment faults once they occur and prevent them from occurring again. As these faults could be non-deterministic, it uses a checkpointing/logging mechanism to capture the execution in an event log, if the execution results in a fault. The framework then applies appropriate environment modifications by altering the event log and replays the execution using the altered log to try and avoid the fault. An environmental change that successfully avoids the fault is recorded, which is then referred to and applied by all future executions of this application to prevent the fault from occurring again. In general, modifications to the environment do not affect the correctness of the application and make the application available immediately instead of having to wait until the actual bug in the program can be fixed. In the remainder of this section these techniques and the framework are discussed in more detail.

Checkpointing/Logging/Replay was already discussed in the previous chapter where it was used for deterministic replay and execution reduction. Previous techniques have used checkpointing/logging to deterministically replay [41, 95] shared memory programs and also in recovery [55] of programs from faults. However, the proposed techniques rollback to a previous checkpoint and replay again without modifying the environment. This process of rollback and replay could avoid some non-deterministic bugs but cannot recover from deterministic ones. Also, these schemes cannot avoid the fault from happening in the future. The framework uses a checkpointing/logging scheme for two reasons. First, it is used to capture the faulty execution to allow deterministic replay, even in the case of non-deterministic faults.

Second, it is used to try to avoid the fault by applying environment changes to the faulty region via modifying the event log. This allows replay of the previous execution such that the faulty region now runs under a different environment that can potentially avoid the fault.

Previous work in this area that is the closest and that has inspired this work is the *Rx* [89] system which was designed to help applications recover from faults due to the environment, by removing the "allergen" that caused the bug to manifest. When a fault occurs, the Rx system rolls back the application to a recent checkpoint and executes it under a modified environment. Repeated environment modifications and re-executions are done until the fault is avoided or a time threshold is passed. If the fault is avoided, the execution is resumed. However, Rx suffers from some drawbacks that motivated the work in this chapter. First, for faults whose symptoms are not immediately apparent, such as a wrong output in a file, the execution could have proceeded beyond many checkpoints before it is detected. Rx will then rollback to a checkpoint which could be very far from the fault. Next the application is reexecuted, *not replayed*, with environment changes that could even affect the previously successful regions of execution. Since the proposed framework supports logging, it replays the execution to figure out the exact point at which the fault occurred. This enables focussing on the region of execution where the environment changes must be applied and also not affect the parts of execution that did not contribute to the fault. Second, Rx cannot prevent the fault in future executions whereas the proposed framework can record the environment change that avoided the fault and apply the same change in all future executions to prevent the fault from occurring again.

In the proposed system framework, each application that runs goes through three main phases: *Logging Phase*; *Fault Avoidance Phase*; and *Prevention-Logging Phase*. Figure 6.1 shows the various phases that an application has to go through in the system. The *Logging Phase* corresponds to the original program run during which the checkpointing and logging infrastructure is turned on. This phase produces the

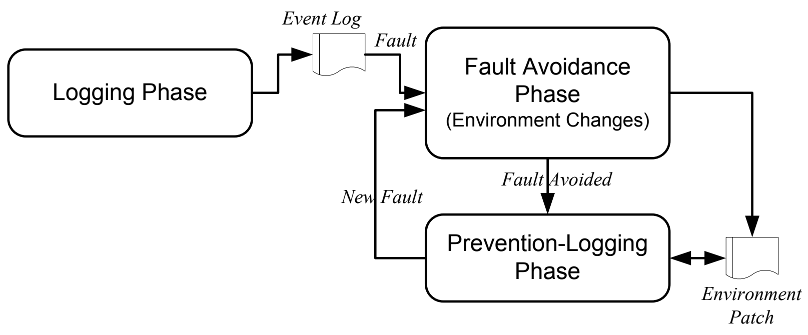


FIGURE 6.1. The various phases that an application goes through in the system.

record of all the events, i.e., the *event log.* The set of logged events can be used to exactly replay the execution when necessary, like when a faulty execution is encountered. Once a fault is detected at any point during the execution or at the end, the application is taken to the *Fault Avoidance Phase.* In this phase, the application is analyzed to correct the fault. If the application was a long-running program like a server, then the clients experience non-availability of the application until the fault is corrected and the application is moved out of this phase. In this phase, the event log of the faulty execution is inspected to detect the nature of the environment bug that manifested in the fault. The system then makes appropriate changes to the event log, which results in altering the environment of the original execution. For example, to avoid atomicity violation errors, a change is made in the order in which threads should be scheduled in the new execution by shuffling the threads in the event log. Now, the program is replayed with the modified event log. This procedure is repeated a few times with different changes each time until the fault disappears. If the fault does disappear, a recording is done of the environment change that avoided the fault and also the region of the application code where the fault occurred in a special log which is called the *Environment Patch.* In the cases where the fault does not disappear, this system cannot be used to prevent them and other techniques for avoiding this fault have to be used. This completes this phase and now the application moves to the *Prevention-Logging Phase.* All future runs of this application are in this phase. In this phase, the patch is referred to when the fault-inducing region is being executed and the appropriate environment settings that will prevent the fault is applied. This ensures that the fault is prevented. Logging is enabled in this phase to capture other faults. The environment patch file is referred to only when an event is logged in this phase. Hence, this merges the overhead of preventing the fault with that of logging the execution. When a new fault occurs, the application moves between the fault avoidance phase and the prevention-logging phase.

Three different types of bugs that can induce environment faults have been looked at. They are *atomicity violation* bugs which are avoided by changing the scheduling decisions, *heap buffer overflow* bugs which are avoided by padding memory requests, and *malformed user request* bugs which are avoided by dropping the request. These bugs were chosen as these could be handled by the Rx system [89] and hence were good candidates for environment bugs. The proposed system has been used on a number of bugs that belong to one of the three types and it has been found that the system can avoid the faults in all the cases.

6.2 Motivating Example

In this section, the approach is motivated using as example, an atomicity violation bug in the `mysql` database server.The remainder of this section describe a known error in a `mysql` version and shows how to apply the proposed approach to the fault.

`mysql` ver. 4.0.12 has an atomicity violation bug [11] which is as follows. This bug has been described in Chapter 2 but is repeated here for convenience. A thread that tries to close and open a new log file atomically in order to flush the previous log gets interrupted just after closing the old log by another thread that does an insert operation into a database. The second thread, hence, does not find any open log files and does not record the insert operation. These logs are used to restore databases and incorrect logs can result in inconsistency. Let us now describe an execution instance

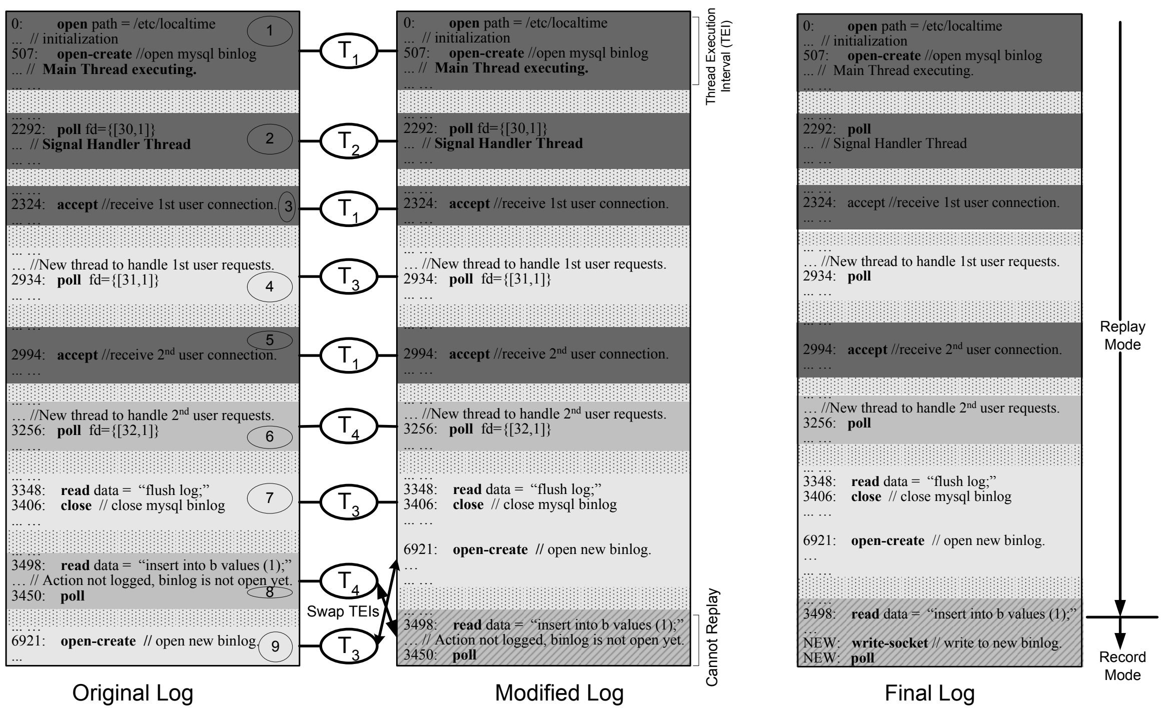
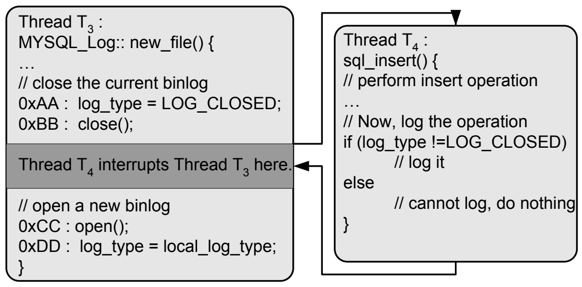

0: open path = /etc/localtime
... // initialization
507: open-create //open mysql binlog
... // Main Thread executing.
2292: poll fd={[30,1]}
... // Signal Handler Thread
2324: accept //receive 1st user connection.
... //New thread to handle 1st user requests.
2934: poll fd={[31,1]}
2994: accept //receive 2nd user connection.
... //New thread to handle 2nd user requests.
3256: poll fd={[32,1]}
3348: read data = "flush log;"
3406: close // close mysql binlog
3498: read data = "insert into b values (1);"
... // Action not logged, binlog is not open yet.
3450: poll
6921: open-create // open new binlog.
NEW: write-socket // write to new binlog.
NEW: poll
Swap TEIs
Thread Execution Interval (TEI)
Cannot Replay
Replay Mode
Record Mode
Original Log
Modified Log
Final Log

FIGURE 6.2. Motivation - `mysql` Atomicity Violation Error. The figure shows the original log corresponding to the error and the modified log where the error is avoided by switching the thread schedules. The final log where the faults has been avoided is also shown. The threads (T_1,. . .,T_4) are shown and the TEIs(1,. . .,9) are marked.

where this fault is exercised and how the system avoids the fault.

Logging Phase. To begin with, the server is run with light weight logging enabled, that is, the events are logged and checkpoints are performed at fixed intervals. Figure 6.2 shows the events recorded in the original event log. T_1, T_2 T_3, and T_4 refer to four unique threads that are created during the execution. The event log also shows the points where a thread is descheduled and another thread is scheduled. A region in the log corresponding to the maximal set of consecutive events from the same thread is referred to as a *thread execution interval* (TEI). The log in Figure 6.2 shows 9 thread execution intervals.

The queries and the activities of the corresponding threads are as follows:



```
Thread T3 :
MYSQL_Log:: new_file() {
…
// close the current binlog
0xAA :  log_type = LOG_CLOSED;
0xBB :  close();
```

```
// open a new binlog
0xCC : open();
0xDD :  log_type = local_log_type;
}
```

```
Thread T4 :
sql_insert() {
// perform insert operation
…
// Now, log the operation
if (log_type !=LOG_CLOSED)
        // log it
else
        // cannot log, do nothing
}
```

FIGURE 6.3. Source code of the `mysql` atomicity violation fault.

- Thread T_1 is the startup thread that handles new connections and creates threads to service requests.
- Thread T_2 is created by T_1 to handle signals.
- Thread T_3 is created by T_1 to handle a user. This user issues a "flush log" command that closes the old `mysql` log and opens a new `mysql` log file
- Thread T_4 is created by T_1 to handle another user. This user does an insertion operation into table 'b'.

Figure 6.3 shows the code that is executed by Threads T_3 and T_4. T_3 is interrupted at the point just after it closes the binlog, corresponding to TEI 7 in the event log. At TEI 8, thread T_4 performs the insert operation but does not find any log open and hence does not record it. At TEI 9, a new bin log is opened but the insert operation is not found to be recorded in any of the logs. Hence, a fault is discovered and the program is taken to the next phase of the framework. The execution could have proceeded much further before this fault is actually detected, like when an administrator runs sanity checks. However, since the event log is present, the execution can be reproduced and the exact point at which the fault occurred could be tracked. Notice that this bug is non-deterministic as the scheduling decisions could be different in another execution instance. Also notice that the log captures the fault successfully.

Fault Avoidance Phase. In this phase, the fault is avoided by applying environment changes to the captured faulty execution. The faulty execution is first replayed, with checks inserted at the start of each TEI, to figure out exactly where the fault occurred. Doing this tells that TEI 8 was faulty as the insert operation is done at this point but is not recorded in the old log. Since the fault is a missing output and the program is multithreaded, it indicates a possible synchronization bug. The system looks for a possible interruption of a running thread at this point. Note that TEI 8 corresponding to thread T_4 interrupted the execution of thread T_3. Swapping TEIs 8 and 9 in the event log makes the interruption go away. The modified event log is shown in Figure 6.2. Now, the execution is replayed with the modified log and as expected, the insert operation is logged in the newly created bin log and the fault is absent.

While replaying from the event log, executing the "insert operation" corresponding to TEI 9 in the original log causes the execution path to change. This is clear from Figure 6.3, where, originally, the condition in the *if* statement evaluated to *false*, it now evaluates to *true*. Hence, the events necessary to replay this is not present in the modified log. So, when this happens, the execution mode is switched from replay to normal execution (record). The final log shows the set of captured events for the correct execution. Notice that there is an entry now in the final event log corresponding to logging the insert operation.

Now that the fault has been avoided, it is now shown how this fault is prevented from happening permanently in the future. Notice that thread T_3 was interrupted by T_4 just when it was performing the close() event of the old bin log, whose PC value is $0xBB$ as shown in Figure 6.3. It is deduced that this must have happened because this region of code, though intended to be atomic, must have been left unprotected. Hence, an entry is added of the form "$< 0xBB >$: Don't Schedule out" in the Environment Patch. It will be shown how this helps in avoiding the fault in the Prevention phase. Now, the server is ready to move to the next phase and start servicing requests normally as the fault has been avoided.

Note that by using the event log, it can be exactly pointed out as to the region of execution where the fault occurs and focus the environment changes. In the Rx system [89], logging is absent and checkpointing alone is done. So, upon such a fault, the system rolls back to a previous checkpoint which could be far away from the faulty region. Now, environment changes are applied starting from the checkpoint and the code is reexecuted, not replayed. This can become ad-hoc as it affects the part of execution that was also successful previously. In the proposed system, since the execution is replayed, the changes do not affect the part that was successful.

Prevention-Logging Phase. In this phase, the application runs normally with logging turned on just like in the logging phase. When each event is being logged, control is transferred from the application to the logging system. At this point, the environment patch is checked to see if the PC of the currently executing event corresponds to a faulty region. For instance, if some thread T_i is executing the piece of code corresponding to PC $0xBB$, the logging system detects that it is a potentially faulty region by looking up the environment patch and also sees that no scheduling must happen at this point. Hence, the priority of the executing thread is raised before the application gains further control. Now, this thread continues executing past this event without being scheduled out and the fault is prevented from happening. The priority of the thread is reset after a predetermined number of events are executed. Since the logging infrastructure is active, any new faults can still be captured in the log. When a new fault occurs, the application moves back to the fault avoidance phase.

6.3 Fault Avoidance and Prevention

In this section some of the details involved in avoiding the three types of environment faults are given. A discussion is presented on how the system discovers the point at which the environment changes must be made to avoid the fault. It is also shown how the system uses the information gathered in avoiding the fault to prevent future occurrences of the fault in each case.

6.3.1 Handling Synchronization Faults

To avoid faults due to synchronization errors among threads, like the example in Figure 6.2, the system first tries to detect the two threads that are involved in the fault. A synchronization error occurs because the execution of a thread, which is called *interuptee*, is interrupted (around an intended atomic region that was not locked) by another thread, which is called *interrupter*. Once the system finds the TEIs of these two threads, the scheduling is modified so that the synchronization error goes away. For instance, in Figure 6.2, thread T_3 is the interuptee and T_4 the interrupter. After the interuptee thread is descheduled, the first thread that executed is the interrupter.

The following is a discussion of the conditions which must be satisfied for a thread to be considered interrupted. Event boundaries at which thread scheduling decisions take place could be synchronous or asynchronous. A thread is interrupted if and only if it is scheduled out after it executed a synchronous event. For example, if a thread after performing a file read event (synchronous) is scheduled out, it is considered interrupted, whereas, a thread which was scheduled out when executing a polling event (asynchronous) is not considered interrupted. The latter case is because the asynchronous event can block the thread for an arbitrarily long interval of time, depending on when the polling is successful, and hence a different thread must be scheduled if execution of the application must proceed. Note that all interruptions do not lead to synchronization errors but this is used as the basis to decide if a synchronization error could have taken place. In Figure 6.2, thread T_4 did interrupt thread T_3 at TEI 8 as the event at which T_3 was scheduled out corresponded to closing a file, which is synchronous. Also at TEI 5 in Figure 6.2, thread T_1 did not interrupt T_3 as polling is an asynchronous event.

In the fault avoidance phase, once the system discovers one of the TEIs where the fault occurred, it uses the analysis presented above to find potential TEIs that could have been interrupted. It then tries to avoid the interruption by letting the interuptee execute further and see if the synchronization error is removed. Again, in the example in Figure 6.2, it is known the fault occured in TEI 8 corresponding to thread T_4. Looking at the log, it is found that an interruption could have happened only in TEIs 7 and 8. Modifying the log at this region avoids the error. Also, note that if the scheduler is changed in the fault avoidance phase, such that all threads execute without interruptions until blocked by an asynchronous event, the error would go away. However, this is not desirable as this would not reveal the exact point at which scheduling was harmful and hence, a patch cannot be obtained to avoid the bug permanently in the future.

6.3.2 Handling Heap Buffer Overflow Faults

When a potential heap buffer overflow fault occurs in a program causing it to crash, the system moves the program to the fault avoidance phase and first detects the heap buffer that had overflowed. This is done as follows. It uses a *hash table* to maintain for each virtual memory address, the EIP of the instruction that performed a memory allocation. It instruments the program at each heap memory allocation instruction that allocates a range of addresses to update the corresponding entries in the hash table with its EIP. It then replays the program from the event log. During execution, for every load and store to a heap address, it checks if the accessed address has a corresponding hash table entry. If an entry is not present, an unallocated address is touched. Now, it looks at neighbouring addresses to see if they have an entry in the hash. If so, the address found, very likely, corresponds to the EIP of the instruction that allocated the heap buffer which overflowed.

Once it has obtained the EIP of the instruction, EIP_{mem}, that allocated the heap buffer, it tries to avoid the fault as follows. It replays the program a second time but pads the memory returned to this heap buffer, by doubling it. If the fault is avoided, it now adds an entry to the environment patch of the form : "$< EIP_{mem} >$: Double Memory". Now, the application is moved to the final phase and all future executions avoid the fault permanently as follows. During a memory allocation call, the EIP of the instruction is checked to see if it matches an entry in the environment patch. If so, the memory to be allocated is padded by doubling it. Notice that only the heap buffer that overflowed previously is padded and this is possible because of the ability to replay the program. The decision to double the memory to be padded is a heuristic based on the faults looked at. For the heap overflow bugs considered, there was an error in calculating the buffer length for some special cases of an input string that overflows the buffer by a small amount and doubling the heap memory was more than sufficient to avoid these faults.

6.3.3 Handling Bad User Request Faults

Faults belonging to this category could be malicious user requests that are intended to expose a bug in the server and not do anything useful otherwise. It could also be a set of user requests that are malformed. These faults usually end up crashing the server by overflowing a stack buffer or even a heap buffer. The strategy, environment modification, to avoid these faults is to ignore such requests. However, this is done as the last resort as dropping requests that are not malicious is a form of *denial of request*. However, this is still better than starving all the users by bringing down the server. Before dropping the request, a check is made to see if these faults can be avoided by padding any overflown heap buffers and modifying thread schedules at regions where there could have been a synchronization error. If all fails, the system drops this request and sees if this will avoid the bug. If so, it saves the EIP of the instruction where this request was accepted, EIP_{read}, along with the user request, req, in the environment patch file as : "$< EIP_{read}, req >$: Drop Request". In the prevention-logging phase, when such a request turns up at this EIP, it is not serviced and the fault is averted. However, a particular request can be fault inducing or not depending on the previous requests. For instance, if request R_i should always be made after request R_j which could crash the server otherwise, then it is not correct to drop R_i every time, that is, in the cases where it was preceded by R_j. Hence, this is taken into account while preventing the request on a per-application basis. This requires manual intervention and the administrator maintaining the patch can look at the request to make an appropriate decision. If it is a "suspicious looking" request, it can be dropped always. If it is a legal request but failed, it could be because it was not preceded with other requests. In this case, the administrator could save the prior window of requests that were made by the user and use this as the pattern to detect before dropping a request. Case studies presented in Section 5.4 and 5.5 further illustrates how this has been implemented.

6.4 System Description

In this section, the implementation of the system is described that incorporates checkpointing/logging and a dynamic instrumentation capability. This system has been used to avoid repeated occurrences of environmental faults in different applications.

Logging and Checkpointing Infrastructure. The *jockey* user level library [96] has been used to perform checkpointing and logging for replay. *jockey* is a very powerful system that works on single and multiple threaded programs and is also

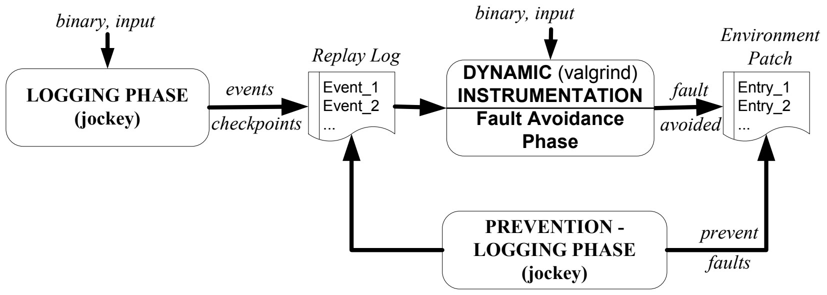


FIGURE 6.4. Implementation of the system showing each step of the framework

very easy to use. During execution, even before the application can run, *jockey* takes control and scans the application binary for system call instructions. It then redirects these calls to a *jockey* handler and lets the application execute. Also, *jockey* provides wrappers for *malloc* calls made by programs. That is, all *malloc* calls in the application are trapped by *jockey* and then redirected to a *jockey* handler. In the handler function, the original *malloc* is called again. During system calls, *jockey* logs events, makes scheduling decisions, creates checkpoints, etc. Scheduling of the user-level threads can be controlled by *jockey* because it uses its own thread libraries and any current thread is descheduled only at a system call boundary. Currently, the *jockey* thread library makes scheduling decisions only while executing one of a small set of system calls. *jockey* has been modified to potentially allow scheduling during any system call and also allow changing the priority of threads. The former is done to increase the number of possible thread schedules in a multithreaded program and expose some of the synchronization bugs in the programs used while the latter is necessary to prevent descheduling of an executing thread at a particular point. Checkpointing is achieved by retrieving the layout of the application's virtual space and dumping all virtual memory segments that belong to the application. To summarize, *jockey* related work is performed only during a system call and otherwise, the application executes as though it was unaware of *jockey*.

The *jockey* system has two modes, recording and replay. In the logging and prevention-logging phases, *jockey* is used in the record mode where it does the work of logging the execution. In the fault-avoidance phase, after the environment change is decided, *jockey* is run in the replay mode to detect if the fault has disappeared. Also, *jockey* has been modified to be able to switch dynamically from the record to the replay mode. This is necessary because the replayed execution might take a different path at the point where the bug happened previously. At this point, the event log is not valid as it no longer contains any information on the new events that are encountered by the application.

jockey has been modified to allow the passing of the *EIP* of each system call and *malloc* instruction in the application as an argument to the *jockey* system call and *malloc* handler, respectively. This information will be used by *jockey* in the prevention-logging phase to determine if an environment change needs to be applied for an event. At system call handlers, the environment patch is consulted to check if scheduling decisions can be made or if a user request needs to be dropped. At *malloc* handlers, the patch is checked to see if the memory needs to be padded.

Dynamic Instrumentation Engine. The dynamic instrumentation tool is used in the fault avoidance phase. It is used to instrument the application and control its execution during the fault avoidance. For example, it is used to insert code in the application so that during the execution it can check if any interesting events have taken place, like, the occurrence of the fault. In the motivating example shown in Figure 6.2, the execution is paused at the end of each TEI to check if the fault has occurred during replay. It is also used in the context of heap overflow bugs to help detect the instruction that allocated the heap buffer which eventually overflowed, according to the procedure discussed in Section 3.2.

To perform dynamic instrumentation, the *valgrind* [80] system has been used which can handle x86 binaries. Its first job is to be able to replay the program according to the log generated by *jockey*. For multithreaded programs, the scheduler decisions are the most important events that have to be replayed. The schedules are replayed as follows. It is known that scheduling decisions in the original program are made only at system call sites by *jockey*'s thread library. When logging the schedule, the system also logs the number of system calls that the thread executed since it was scheduled and before it was descheduled. In *valgrind*, for a thread that is currently executing, the number of system calls executed is used to decide when to deschedule the thread. When that system call is reached (*valgrind* has event handlers that are called before and after a system call and this is used to count system calls), the scheduler is forced to deschedule this thread and switch to the appropriate thread. Hence, it can be guaranteed that the threads will be scheduled according to the event log. File contents are restored so that system calls performing opening and closing of files can be replayed exactly. There are some system calls for which *jockey* saves the contents of the original run. For example, in a server program, if a client makes a connection request, the contents of the *socket-read* system call are saved in the *jockey* log. During replay, when the system call *socket-read* is to be executed, *jockey* will return the saved contents instead of executing the system call. Something similar is done in *valgrind* when replaying the program. When the *socket-read* system call is reached, the system call is not performed instead contents are returned which are saved in the *jockey* log.

Given a logging infrastructure to be used in all the phases and an instrumentation mechanism that is used in the fault avoidance phase, it is possible to successfully capture, avoid and prevent environment faults. The next section contains case studies of faults from each type and it is shown how each of them was avoided.

6.5 Case Studies

In this section five case studies of environment bugs from real world programs are described. The nature of the fault that occurs due to the bug and discuss how the system successfully averts it is explained. The bug is first described that causes the fault to occur and then shown how the environment change can avoid it. Also the patch used to prevent this fault from happening in the future is shown.

6.5.1 Atomicity Violation Fault in `mysql`

According to the bug report [12], `mysql` ver. 3.23.56 has an atomicity violation error which is as follows. This bug has been described in section 2.2.2 in Chapter 2 and is described again here for convenience. For some table 't' in the database, when one thread does a row delete from it and another thread does an insert into it in quick succession, though the operations take place in the order they are called, they are logged in the `mysql` binlog as done in the reverse order. The `mysql` binlog does not reflect the true sequence of operations on the same table and hence it is inconsistent with the state of the table as shown below.

—– Log File —–

*SET TIMESTAMP=*1151980120*;*

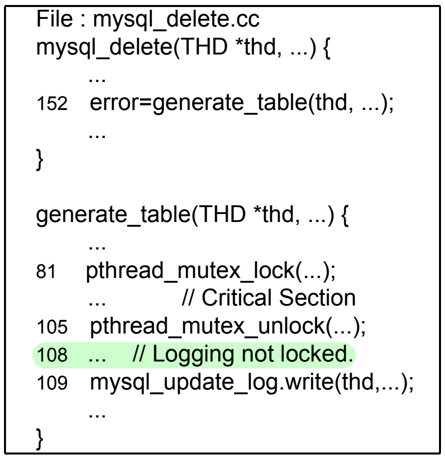
insert into b values (1);

*SET TIMESTAMP=*1151980107*;*

delete from b;

—– End of Log File —–

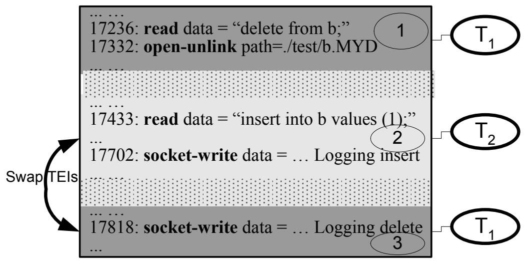
Notice that although the delete operation is done first it gets logged after the insert operation. The reason is that line 109 in Figure 6.5 which performs the write to the binlog is not inside the critical section. So, the thread corresponding to the insert operation gets scheduled before this point and hence, this inconsistency occurs. Figure 6.6 shows the event log corresponding to the faulty execution. When the system replays the program using the log, it detects that TEIs 1 and 2 are directly involved in the fault as the delete and insert operations take place during these points. It also detects that TEI 2 interfered with the execution of thread T_1. The execution



```
File : mysql_delete.cc
mysql_delete(THD *thd, ...) {
       ...
152   error=generate_table(thd, ...);
       ...
}

generate_table(THD *thd, ...) {
       ...
81    pthread_mutex_lock(...);
       ...          // Critical Section
105   pthread_mutex_unlock(...);
108   ...    // Logging not locked.
109   mysql_update_log.write(thd,...);
       ...
}
```

FIGURE 6.5. `mysql` 3.23.56-Source code for atomicity violation fault.

of thread T_1 is hence extended by swapping TEIs 2 and 3 and the fault is avoided. It also notes the PC at line 108, which is $0x81023AC$, in the environment patch with the command not to schedule at this point. The patch is hence "$< 0x81023AC$: Don't schedule>". In the prevention-logging phase, when the execution reaches this point, the thread's priority is raised so that it does not get descheduled. After applying the patch, as a sanity check, the server was continued to execute again and the same sequence of operations was performed to ensure that the fault was indeed prevented.

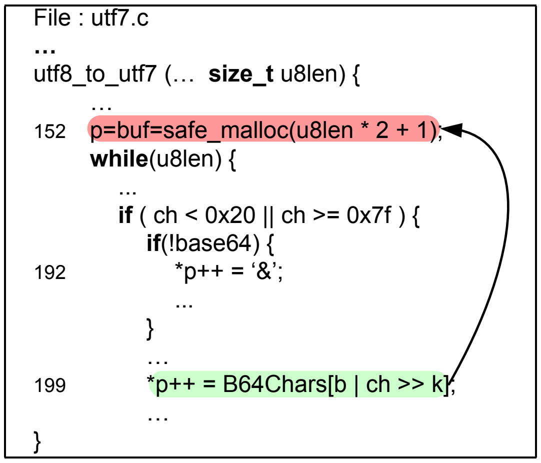


```
... ...
17236: read data = "delete from b;"
17332: open-unlink path=./test/b.MYD
... ...

... ...
17433: read data = "insert into b values (1);"
...
17702: socket-write data = … Logging insert
... ...

... ...
17818: socket-write data = … Logging delete
...
```

FIGURE 6.6. `mysql` 3.23.56-Event Log for atomicity violation fault.

6.5.2 Heap Buffer Overflow Fault in `mutt`

According to the bug report[9], `mutt` version 1.4 has a known memory bug which is as follows. This bug has been described in section 2.1.1 in Chapter 2 and is described again here for convenience. The Mutt Mail User Agent (MUA) has support for accessing remote mailboxes through the IMAP protocol. When `mutt` has to convert the name of the folder from its internal UTF-8 representation to UTF-7 it calls the function *utf8_to_utf7* in module *imap/utf7.c*. When this function does the conversion, it miscalculates the length of the output string, line number 152 in Figure 6.7. To form a faulty execution, `mutt` is executed for some time and then supplied a UTF-8 folder name that contains some special characters. The heap buffer is overflowed and a segmentation fault is flagged. The *jockey* event log captures all the events necessary to replay the fault. The application is then taken to the fault avoidance phase.



```
File : utf7.c
...
utf8_to_utf7 (... size_t u8len) {
        ...
152     p=buf=safe_malloc(u8len * 2 + 1);
        while(u8len) {
            ...
            if ( ch < 0x20 || ch >= 0x7f ) {
                if(!base64) {
192                 *p++ = '&';
                    ...
                }
                ...
199             *p++ = B64Chars[b | ch >> k];
                ...
}
```

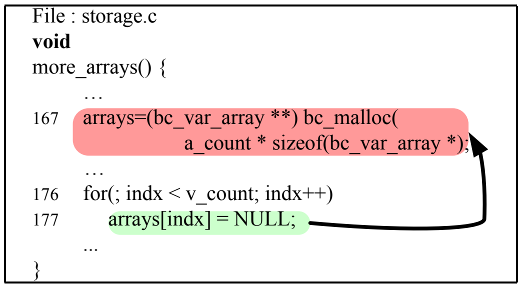
FIGURE 6.7. `mutt` 1.4.2.1i-Source code for heap buffer(p) overflow fault.

The heap buffer overflow is detected at line number 199 in Figure 6.7 using *valgrind*. The allocation point for this heap buffer is also detected at line number 152. The system then captures the PC of the malloc call at this point which is $0x80A9FBA$. The program is replayed doubling the memory allocated at this point and the fault goes away. This is recorded in the environment patch with the entry

: "$< 0x80A9FBA :$ Pad Allocation>" and then the program execution continues with the fault avoided. The application is the run for some time and again the fault-inducing request is presented. The *jockey*'s malloc wrapper successfully makes the environment change and prevents the fault from happening again.

6.5.3 Heap Overflow in `bc`

`bc` [1] is a numeric processing language that supports arbitrary precision numbers. It is generally distributed along with the Linux operating system and is a part of the GNU project. `bc`-1.06 was used for the case study. This version has a known heap overflow error. In [67] the bug that is triggered in `bc` is described. The code corresponding to the error is shown in Figure 6.8. The heap array *arrays*, declared at line number 167, overflows at line 177. Upon capturing the faulty execution, the



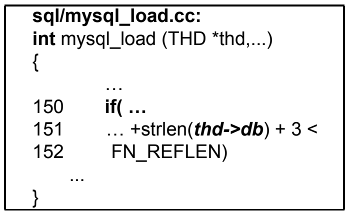
```
File : storage.c
void
more_arrays() {
        …
167     arrays=(bc_var_array **) bc_malloc(
                    a_count * sizeof(bc_var_array *));
        …
176     for(; indx < v_count; indx++)
177         arrays[indx] = NULL;
        ...
}
```

FIGURE 6.8. `bc`-1.06 Source code for heap overflow fault.

system detects the allocation point and doubles the memory. The fault disappears. As before, it obtains the PC value and records it in the environment patch with the entry : "$< 0x804CDA3 :$ Pad Allocation>". For the same input, this fault will not reappear. However, the patch is really temporary as the root cause of the bug is that the variable *a_count* is used to declare the array but the variable *v_count* is used to initialize it. Doubling the memory worked for the input presented because *v_count* was not much larger than *a_count* but this is not always the case. It is likely that the program will crash at this point again on other inputs, in which case, the memory to be padded must be increased to a large value.

6.5.4 Bad User Request Fault in `mysql`

According to [14], `mysql` version 3.23.56, has a memory error which is as follows. When a thread tries to load data into a table using the "load" command, without explicitly connecting to the database, the field that stores the database name is accessed without checking for the *NULL* value (line 151 in Figure 6.9). This causes the server to crash at this point. After capturing the log corresponding to this fault



```
sql/mysql_load.cc:
int mysql_load (THD *thd,...)
{
        ...
150     if( ...
151     ... +strlen(thd->db) + 3 <
152      FN_REFLEN)
     ...
}
```

FIGURE 6.9. `mysql` 3.23.56-Source code for malformed user request fault.

it is detected that there is an illegal memory access. A possible heap overflow fault is searched for and the system is not successful as it cannot find any heap buffer that overflowed. This is because the field containing the database was not allocated and contained a NULL value. The system does not detect any interfering threads either around this point. So, it traces trace back to the most recent read system call looking for a request. It finds the "load" command, which when dropped, clearly avoids the fault. However, the command is a legitimate `mysql` command with no special characters. Hence, always dropping this request is a bad idea. The only pattern noticed is that this is the first command from this user. Hence, it forms a pattern where it drops this request if it is the first request after the connection. Now, in the prevention-logging phase it saves all the requests corresponding to each user and if this pattern is detected, drops it. The entry used to achieve this is : "$< 0x80CCAFD$, *pattern-string* : Drop>", where the pattern string is a connection request followed by this "load" command. Now, when this command is presented as the first request, it will not be serviced and hence will not bring down the server. If more faults occurs at the same point due to other malformed requests, many patterns will exist. In this case, a match must be done with all patterns and a request that matches any one of the patterns must be dropped.

6.5.5 Bad User Request Fault in `pine`

According to [15], `pine` ver.4.44 has a bug that when triggered can overflow a heap buffer causing a crash. This can occur when `pine` processes the "From" field of email headers. Certain special characters in the header can cause the bug. Figure 6.10 shows the source code where the bug is present. The heap buffer *dest* overflows in line 260 in function *rfc822_cat()* as the amount of memory allocated to it is miscalculated in line 7269 in function *est_size()*. After capturing the event log corresponding to this

```
File : bldaddr.c
int est_size(a) {
        …
7269    cnt += …
        …
        return(max(cnt,50));
}

File : rfc822.c
void rfc822_cat (char *dest, …) {
        …
        dest += strlen(dest);
        *dest++ = '"';
        …
        for(;s = strpbrk (src,"\\\""); …) {
            strncpy (dest, …);
            dest += i;
260         *dest++ = '\\';
            *dest++ = *s;
        }
}
```

FIGURE 6.10. `pine` 4.44-Source code for bad user request fault resulting in heap overflow.

fault, the system first tries to avoid the fault by padding memory. It tracks the heap buffer that was overflown and double the memory at this point but the bug does not disappear. Hence, it detects the request that caused the bug to occur and observes that the request is unusual as it is full of special characters in it. Hence, it decides to drop such requests and adds an entry to the environment patch with the contents, “<$0x3A976422$, *pattern-string* : Drop>”, where the pattern string is the string of special characters that caused the fault.

6.6 Experiments

Table 6.1. Benchmarks and the bugs used in the Experiments.

Program	Description	LOC	Description of bugs used
`mysql`	Database	508 K	a) Atomicity bug[11]
	(ver. 4.0.12)		`mysql-1`
	(ver. 3.23.56)		b) Atomicity bug[12]
			`mysql-2`
	(ver. 4.00)		c) Atomicity bug [13]
			`mysql-3`
	(ver. 4.00)		d) Bad Req. bug [14]
			`mysql-4`
`pine`	Mail client	212 K	a) Bad Req. bug [15]
	(ver. 4.44)		`pine-1`
	(ver. 4.44)		b) Bad Req. bug [16]
			`pine-2`
`mutt`	Mail client	454 K	a) Heap Overflow [9]
	(ver. 4.44)		`mutt-1`
`bc`	Interactive	14 K	a) Heap Overflow [67]
	Calculator		`bc-1`
	(ver. 1.06)		b) Heap Overflow [67]
			`bc-2`

Table 6.1 shows the list of buggy versions of programs that have been used to evaluate the system. Each of the bugs belongs to one of the three possible types of

Bug	Logging Phase			Prevention-Logging Phase		
	Orig. (secs.)	Logged (secs.)	Logged / Orig.	Logged (secs.)	Prevention (secs.)	Prev. / Logged
`mysql`-1	15.5	15.9	1.03	16.0	16.1	1.01
`mysql`-2	7.8	8.0	1.03	8.0	8.5	1.06
`mysql`-3	7.2	7.4	1.04	8.7	8.9	1.02
`mysql`-4	7.9	8.1	1.02	8.7	9.0	1.03
`pine`-1	6.1	6.8	1.11	8.1	8.4	1.04
`pine`-2	4.3	4.9	1.14	6.1	6.2	1.02
`mutt`	6.7	7.9	1.18	9.0	9.2	1.02
`bc`-1	6.5	7.4	1.14	10.1	10.1	1.0
`bc`-2	4.3	4.5	1.05	6.2	6.3	1.02

Bug	Fault Avoidance Phase			
	Trials	*valgrind* (secs.)	*jockey* (secs.)	Environment Change
`mysql`-1	1	116	15.8	Scheduler
`mysql`-2	1	58	8.0	Scheduler
`mysql`-3	1	59	7.3	Scheduler
`mysql`-4	3	682	22.8	Ignore Req.
`pine`-1	2	314	13.2	Ignore Req.
`pine`-2	2	262	9.0	Ignore Req.
`mutt`	1	197	7.7	Pad Mem.
`bc`-1	1	285	7.4	Pad Mem.
`bc`-2	1	190	6.2	Pad Mem.

Table 6.2. Overheads involved in each of the three phases - logging, avoidance and prevention-logging.

environment faults that are looked at. The buggy version of each program was taken and an execution was formed that runs for some time, between 4 and 15 seconds, and then the fault was introduced. For example, in m`mysql`, a few clients were created that processed a set of standard requests from each client and then the fault was triggered by issuing the fault inducing request. Since the executions were not too long, checkpointing was not triggered. After patching the fault by applying the appropriate environment change, the application was run again for some time and the fault was introduced as before. It was ensured that the fault is indeed avoided by continuing execution beyond the point for a couple of seconds before terminating it. The various experiments conducted are described.

Logging Phase. The overheads involved in logging the execution was measured during this phase. In Table 6.2, under Logging Phase, the running time of the application without logging (Original) and with logging (Logged) is shown. The overhead of logging, shown under *Logged/Orig.*, is between 2% and 18% and this shows that the logging mechanism is lightweight enough to be run along with the application at all times.

Fault Avoidance Phase. The costs incurred in the fault avoidance phase where environmental changes are applied to avoid the bug is shown. The data for this is shown under *Fault Avoidance Phase* in Table 6.2. The number of tries to avoid each bug, under the column *Trials*, is also shown where each try corresponds to a different environmental change. All faults that were triggered by malformed requests needed more than one trial as the system first checked if it could fix the fault by padding memory or changing thread schedules. All failed, and hence dropped the request. The column *jockey* shows the total time spent to replay the program using *jockey*, with the environment changed, to detect if the fault was avoided. The data shows that the *jockey* time for one trial is almost equal to the original time. The column *valgrind* shows the time taken to replay the program in *valgrind* to detect the regions corresponding to the fault and the time taken to perform analysis, like detecting the allocation point given a heap overflow. This incurs a slowdown of 7x-44x per trial. Note that this cost is incurred only in this phase due to the expensive analysis that is performed using *valgrind*. This overhead will not be present in the prevention-logging phase when the application is running normally. The column *Environment Change* shows the patch that avoided the bug and used to prevent it from occurring again.

Prevention-Logging Phase. Finally, the overheads of performing logging and preventing faults are shown, that are incurred in this phase. Table 6.2 shows these costs under Prevention phase. The column *Logged* shows the overhead of logging the execution in this phase with the bug fixed in the source code and the prevention mechanism turned off. The column *Prevention* shows the time taken to perform logging and prevention on the application with the bug present. The additional overhead of preventing the fault beyond the logging overhead is shown under the column *Prev./Logged*. The overhead, which ranges from 0% to 6%, is low and is due to the fact that the system could successfully merge the operation of checking the environment patch with the logging. The combined overhead of the logging with prevention mechanism is between 2% and 19% and is low enough that it can be run alongside the application always.

6.7 Summary

This chapter presented a scheme that uses logging and environment patching to capture and avoid environment faults as and when they occur and prevent them from occurring again. Case studies are presented to show that the scheme can be successful against three types of environment faults and this has been verified on nine known bugs in real-world applications. Experimental data shows that the overhead of the logging and prevention mechanism is low enough, 2% to 19%, to justify it being run alongside the application at all times.

Chapter 7
Related Work

7.1 Fault Location

The dissertation of Zhang [113] "*Fault Location Via Precise Dynamic Slicing*" focuses on techniques for *fault location using dynamic slicing* for single-threaded programs. It identifies novel slicing criteria which are used to generate dynamic slices that are highly effective in capturing the faulty code. According to [113], the work on fault location can be divided into four categories, which is briefly discussed here.

First, for approaches based on *dynamic slices* for fault location, the work by Korel and Laski [59] introduced dynamic slicing as an effective aid to debugging programs. Agrawal et al. [21] proposed a technique for *fault localization* by subtracting a single correct execution trace of a program from a single failed execution trace. Other works [19, 57, 60, 86, 102] have also looked at slicing approaches for fault location. An approach that combines dynamic slicing with dynamic path conditions in dependence graphs [47] has been proposed that can not only reveal if a dependence has been exercised but also why which is used to capture the erroneous statement and fix the fault. Chen and Cheung [31] propose a technique to reduce the number of statements in a backward slice of an erroneous output by computing differences between the backward slices of correct and faulty statements. Second, there is also a large amount of work [40, 48, 56, 67, 70, 90, 107] that has looked at statistical approaches for fault location whose goal is to obtain the likelihood of a statement being faulty based on numerous runs and then rank the statements accordingly. Third, there has also been work [34, 50, 52, 112] on fault localization by identifying the program state that was critical to the failure by using a successful and failed run. Finally, there has been work [22, 29, 38, 39, 42, 43, 51, 53, 65, 71, 74, 87, 106] on using static analysis for fault location where first a correct program model is constructed according to the specifications and static analysis is used to verify conformance of the program with the given model. Recently, a scheme [75] to systematically explore multithreaded program executions using model checking has been proposed which can reduce the search overhead by prioritizing executions of a multithreaded program based on the number of context switched.

7.2 Slicing for Multithreaded Programs

The paper by Xu et al. [108] gives a brief survey of program slicing that also includes the works for multithreaded programs. Some of the relevant approaches are discussed here.

For multithreaded programs, there have been approaches to compute both the static and the dynamic slices. One of the early works [33] extend the notion of slicing for concurrent programs. It presents a graph based approach to computing the slice. Three new forms of dependences are defined and used : *selection, synchronization and communication dependences* that can occur in distributed programs. These dependences are nothing but inter and intra-process control and data dependences. This method computes inaccurate slices. The work by Krinke et al. [62] defines a static slice for a multithreaded program. Apart from the traditional data and control dependences, *interference dependences* are considered which occur between two different threads and could manifest as RAW, WAR or WAW at run-time. This algorithm does not consider synchronization and hence is not general. Since the schemes are static, conservative effects could lead to potentially large slices.

Duesterwald et al. [37] propose a parallel algorithm to compute dynamic slices of parallel programs. Since the algorithm is developed for distributed programs, communication dependences are added at run-time to capture the interactions between the distributed processes. The dynamic slicing scheme is targeted towards producing an *executable slice* of the statement that produced the faulty result. The slices are computed based on the dynamic dependence graph (DDG) representation, though the control dependences are determined statically. Also, other works [61, 58] have been proposed to compute dynamic slices of distributed programs that show how to do *interprocedural dynamic slicing* and perform *forward computation* of dynamic slices. The work proposed in this dissertation however is how to inspect the dynamic slice of the faulty execution in order to determine the root cause of the error. The proposed technique is more general and applies to shared memory concurrent programs.

7.3 Program Tracing

A control flow trace captures the complete path followed by a program during an execution. It is represented as a sequence of basic block ids (or Ball-Larus path ids [24]) visited during the program execution. These traces have been analyzed to determine execution frequencies of shorter program paths [64]. Thus, hot paths in the program can be identified and this knowledge has been used to perform *path sensitive instruction scheduling* and *optimization* by compiler researchers [23, 26, 45, 111] and *path prediction* and *instruction fetching* by architecture researchers [54]. Larus has demonstrated that complete control flow traces of reasonably long program executions can be collected and stored by developing the compressed representation called the *whole program path* (WPP) [64].

Dependence (data and control) traces have also been used in a variety of applications. Compiler researchers have used these profiles for performing *data speculative optimizations for Itanium* [68, 69], speculative optimizations [32] and computation of dynamic slices [103, 20, 59, 116]. The latter have been used for software *debugging* [19, 60, 117, 114], *testing* [57] and providing security. Architecture researchers have used slicing to study the characteristics of *performance degrading load instructions* [121], *thread creation using slicing* [66], and *studying instruction isomorphism* [98].

Efficient representations have been proposed when traces are *held in memory* for analysis such as t he timestamped representations of control flow traces [119] and dependence traces [116]. Also, the *whole execution trace* [115] has been proposed which can capture the control-flow, data dependence, value and address profile of a program's execution. An *instruction level tracing* framework is proposed by Bhansali et al. [25]. These state-of-the-art techniques [115, 25] can generate traces to achieve a space efficiency of 0.1-4 bits per instruction. Recent work [120] has shown how to capture a complete profile of a program's control flow, memory reference, and dependence information by exploiting the fact that most of the information can be retrieved by recording the register value changes and using this in conjunction with the control flow trace. Also, recently the ONTRAC [76] system is proposed which computes the *dynamic depdendences online* without generating a trace. Since the trace generated is stored in a buffer in memory it limits the length of the execution that can be traced.

7.4 Checkpointing, Logging, Replay, and Tracing

Checkpointing/logging/replaying was invented to facilitate debugging parallel and distributed programs [85, 104]. It quickly gained popularity in general application debugging [92, 93]. A lot of research has been carried out on how to reduce its cost [99, 82] and improve its usability [96]. It has been used in the *deterministic replay* [41, 95] of shared memory programs and also in recovery [55]. However, most of the existing checkpointing techniques focus on how to faithfully replay an execution. They do not discuss what to do with the replayed execution and simply suggest that the replayed execution can be debugged with general debuggers such as gdb. However, these debuggers are usually less powerful than tracing based tools. This dissertation uses checkpointing/logging schemes for two different purposes. It is used to replay the program with a reduced log to generate smaller traces in the execution reduction system. It is also used to perturb the program execution for fault avoidance.

The prior work, *Execution Fast Forwarding* (EFF) [118] system, also performs a form of Execution Reduction by integrating checkpointing with fine-grained tracing. It is based on the idea that often a fault is triggered by a certain input. By filtering the inputs to find the fault triggering input, the fault can be reproduced. Tracing can then be applied to the smaller program run corresponding to the triggering input. However, the *Execution Reduction* (ER) system proposed in this dissertation is much more general than the EFF system. In particular, the advantages of ER over EFF include the following:

- The ER system is designed to handle multithreaded applications while the EFF system was designed for single threaded applications. One of the key contributions of the ER system is the dynamic algorithm that is proposed for efficiently identifying the interthread dependences. The EFF system does not address this issue as it does not consider multithreaded applications.

- In the EFF system, the execution reduction is achieved by exploiting information collected using static analysis. The disadvantage of using static analysis is that it is conservative and hence dynamic opportunities for achieving execution reduction cannot be exploited. In particular, if static dependences do not manifest themselves at runtime, this information can be exploited for execution reduction in the ER system but not in the EFF system.

- The ER system does not necessarily trace the entire execution that is replayed. In contrast, the EFF system traces the entire execution that is replayed. Therefore, in the ER system, the reduction in tracing is not limited by the amount of execution reduction achieved.

- Finally, input filtering that is used as the basis of execution reduction in EFF is the special case of execution reduction achieved by the ER system. ER system can handle a variety of situations, including those where input filtering is applicable.

In summary, the attractive features of the ER system are that it is more general and more effective than the EFF system.

There has been some recent work that propose designs of specialized hardware to limit the overhead of checkpointing/logging [109, 78] and efficiently recording concurrent shared memory dependences when running on a multiprocessor [110, 77]. These systems aim to minimize the overhead incurred while logging an execution for replay. This work is orthogonal to the proposed work in this dissertation as these systems could be used to improve the performance logging the execution to replay deterministically.

7.5 Fault Avoidance

Previous work in this area that is the closest to and that has inspired the fault avoidance work proposed in this dissertation is the *Rx* [89] system which was designed to help applications recover from faults due to the environment, by removing the "allergen" that caused the bug to manifest. When a fault occurs, the Rx system rolls back the application to a recent checkpoint and executes it under a modified environment. Repeated environment modifications and re-executions are done until the fault is avoided or a time threshold is passed. If the fault is avoided, the execution is resumed. However, Rx suffers from some drawbacks. First, for faults whose symptoms are not immediately apparent, such as a wrong output in a file, the execution could have proceeded beyond many checkpoints before it is detected. Rx will then rollback to a checkpoint which could be very far from the fault. Now, the application will be reexecuted, *not replayed*, with environment changes that could even affect the previously successful regions of execution.

Avio [73] is a technique to detect *atomicity violation* bugs in programs. The main idea of the technique is to use a number of correct runs, with different interleavings in each run, of the application on the same input and discover atomic regions of the program. Once the proposed fault avoidance system in this dissertation has detected the point at which atomicity is possibly violated, it can pass this information to the avio system that can use it to detect if an invariant exists.

Failure-Oblivious computing [91] is a technique that bypasses faults in applications by altering the behaviour of the application when it detects accesses to unallocated memory. It manufactures values for incorrect reads and ignores illegal writes to let the application continue further execution without crashing. This approach needs modifications to the application and the correctness of the application cannot be guaranteed.

A number of *dynamic fault detection* techniques [35, 36, 49, 88] exist that instrument the program to check for illegal memory accesses, deadlocks and data races at run-time and also have reasonable overhead. Such techniques can be used in the proposed system to flag a fault when it occurs.

Chapter 8
Conclusion

8.1 Contributions

This dissertation makes contributions in the area of fault location and avoidance for multithreaded programs. It shows how dynamic slices can be used for fault location in multithreaded programs. Additionally, it is shown how dynamic slices can be used to track down faults due to data races in multithreaded programs. To allow dynamic slicing to scale for large executions, this dissertation shows that the generated control flow and data dependence traces from a faulty execution can be stored very compactly on disk; these traces are used to construct the dynamic slices. It also presents techniques to reduce the sizes of the generated traces in long-running multithreaded programs. Finally, this dissertation presents a technique for fault avoidance in programs due to a certain class of faults referred to as environment faults. Specifically, the dissertation answers the following questions.

Q1 : Can dynamic slices be used for locating faults including data races in multithreaded programs ? This dissertation shows that dynamic slicing can indeed be used for fault location in multithreaded programs. It shows how the slices are represented and presents case studies to show how they are inspected to locate the root cause of the fault. This dissertation also shows that if additional dependences in the form of WAR and WAW are considered then dynamic slices can also be used to analyze data races. It is also presents optimizations that can be used to effectively capture only that set of WAW and WAR dependences that can lead to a data race. These optimizations can reduce the number of WAW and WAR dependences to be captured by upto 4 orders of magnitude.

Q2 : Can the generated traces be stored on disk efficiently ? The *Extended Control Flow Trace* (eCF) representation is described in this dissertation which is a *unified* representation of control flow and data dependence traces and leads to very compact trace sizes to be efficiently stored on disk. The eCF representation captures both the control flow and the data dependence history of a program's execution. In this representation, the data dependences are not captured by an explicit representation. Instead, data dependences are embedded implicitly in the control flow trace. This representation of dynamic data dependences is motivated by the observation that all dynamic register dependences can be recovered from the control flow trace. To capture the remainder of the dynamic data dependences, i.e., memory dependences, program transformations are presented that introduce *disambiguation checks* and whose control flow signatures capture the results of these checks. The resulting extended control flow trace produced enables the recovery of otherwise irrecoverable memory dependences. Thus, this approach replaces the combination of a control flow trace and data dependence trace with a single *extended control flow trace*, which can be compressed well to produce the *extended whole program path* (eWPP) representation. Experimental evidence which shows that the representation produces traces that can be stored in 24 % (30 %) of the space required to store the control flow and address trace when compressed using Sequitur (VPC).

Q3 : Can the generated traces of long-running multithreaded program executions be shortened ? This dissertation describes the *execution reduction system* that can effectively combine checkpointing and tracing in order to generate traces for long-running multithreaded programs. The system uses dynamic techniques for eliminating the execution of irrelevant threads and irrelevant thread execution intervals from the final replay phase that collects traces. Further, it also eliminates unnecessary tracing during the replaying of relevant threads and thread execution intervals. The combined effect of the above approach is that the tracing overhead and the amount of trace data collected is greatly reduced. Most importantly, to make the above scheme work, a three stage scheme is proposed for identifying dynamic shared memory dependences between executing threads that is both space and time efficient. Detailed experiments demonstrate the effectiveness of the proposed techniques in reducing the trace sizes by 2 to 5 orders of magnitude.

Q4 : Can faults be avoided when execution must continue ? This dissertation finally presents a scheme that can capture and avoid environment bugs by using a checkpointing/logging system. The presence of logging enables focussing the environment changes on the faulty region. The scheme can also prevent the captured environment bugs from occurring again by modifying the execution environment without having to debug the program. The proposed system can handle three different types of bugs, namely, atomicity violation, heap buffer overflow, and malformed user request. The scheme has been tested on many bugs from real-world programs, which have shown it to be effective.

8.2 Future Work

Locating Faults due to atomicity violation errors. Atomicity violation faults happen in multithreaded programs when a set of operations intended to be atomic are not guarded by a single lock. The fault happens when another executing thread interleaves the set of operations. Atomicity violation errors can still happen in programs that are free of data races. For instance two operations intended to be atomic could be in different critical sections of the program. Hence, although races are absent in the presence of locks the atomicity is still not preserved as the operations have to be in the same critical section to be considered atomic. Hence, work in this direction to locate atomicity violation faults using dynamic slicing in the absence of data races is promising. The *AVIO* technique [73] for detecting atomicity violations discusses these kind of faults in greater detail.

Hardware support for tracing. Processors with many cores are becoming increasingly available. Hence, it is very promising to investigate techniques using support from additional cores to simultaneously perform tracing while the program is running on a dedicated core. This can tremendously boost the performance of tracing and can make it possible to trace on-line. The tracing can be done as follows. A secondary core can be used to perform tracing while the main core is executing the program. The secondary core also runs the program but additionally performs tracing while the primary core is executing the program. However, the challenges involved in such a scheme are that non-deterministic events have to be communicated in some manner from the primary core to the secondary core like thread scheduling events, outcome of system calls, etc. Further, if the program's threads are executing concurrently, the synchronizations have also got to be communicated.

Programs executing on multi-cores. The system developed in this dissertation for tracing and fault avoidance assume that the programs are executing on a single processor system. However, with multicores becoming popular, multiple program threads can concurrently execute on these cores. This requires the schemes for tracking dependences between threads and instructions to be modified. This is because when the multithreaded program is executing on a single processor it is enough to remember the scheduling decisions in order to replay the threads deterministically in the same order. However, when the threads are executing concurrently on multiple processor cores, additional information is required for deterministic replay. For instance, the inter-thread shared memory dependences of the concurrently executing threads must be logged. While the proposed techniques in this paper are not restricted in any manner because of this new scenario, they have to be adapted to be able to work in this setting. This is definitely potential for a lot of future work and is also very promising.

Identifying and avoiding other environment faults. The system developed in this dissertation for fault avoidance handled three types of environment faults. However, there are many other types of environment faults that occur and it would be interesting to study how they can be avoided and patched. One such type of fault not handled by the proposed system is due to freeing of memory in *C* programs using the *free()* command. When the same piece of memory if freed more than once the application can crash. This is an environment fault because the memory handler can be made to ignore duplicate free operations thereby preventing the crash. In this manner, if more environment faults are identified and the suitable patch to try and prevent them from happening again is developed, the system can be extended to handle a large number of fault types thereby making it more robust.

References

[1] Gnu bc url : http://www.gnu.org/software/bc.

[2] http://bugs.mysql.com – change log.

[3] http://bugs.mysql.com/bug.php?id=110.

[4] http://bugs.mysql.com/bug.php?id=169.

[5] http://en.wikipedia.org/wiki/y2k.

[6] http://prozilla.genesys.ro/?p=news.

[7] http://www.securityfocus.com/bid/12635.

[8] http://www.securityfocus.com/bid/13059.

[9] Mutt buffer overflow. http://www.securiteam.com/unixfocus/5fp0t0u9fu.html.

[10] Mutt url : www.mutt.org.

[11] Mysql atomicity violation-1 : http://bugs.mysql.com/bug.php?id=791.

[12] Mysql atomicity violation-2 : http://bugs.mysql.com/bug.php?id=169.

[13] Mysql atomicity violation-3 : http://bugs.mysql.com/bug.php?id=6678.

[14] Mysql load database fault : http://bugs.mysql.com/bug.php?id=110.

[15] Pine heap buffer overflow : http://www.securityfocus.com/bid/6120.

[16] Pine stack overflow : http://www.xatrix.org/advisory.php?s=7408.

[17] www.mysql.org.

[18] http://www.nist.gov/public_affairs/releases/n01-10.html. 2002.

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

[20] 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.

[21] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 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.

[22] 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.

[23] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 72–84, New York, NY, USA, 1998. ACM Press.

[24] 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.

[25] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM Press.

[26] 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.

[27] Martin Burtscher. Vpc3: a fast and effective trace-compression algorithm. In *SIGMETRICS : International Conference on Measurement and Modeling of Computer Systems*, pages 167–176, 2004.

[28] Martin Burtscher and Metha Jeeradit. Compressing extended program traces using value predictors. In *IEEE PACT : International Conference on Parallel Architectures and Compilation Techniques*, pages 159–168, 2003.

[29] 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.

[30] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and*

Networks (formerly FTCS-30 and DCCA-8), pages 97–106, Washington, DC, USA, 2000. IEEE Computer Society.

[31] 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.

[32] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data dependence profiling for speculative optimizations. In *CC : International Conference on Compiler Construction*, pages 57–72, 2004.

[33] Jingde Cheng. Slicing concurrent programs - a graph-theoretical approach. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 223–240, London, UK, 1993. Springer-Verlag.

[34] 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.

[35] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, 2003.

[36] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[37] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Distributed slicing and partial re-execution for distributed programs. In *LCPC : The International Workshop on Languages and Compilers for Parallel Computing*, pages 497–511, 1992.

[38] 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.

[39] 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.

[40] 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.

[41] Yishai A. Feldman and Haim Schneider. Simulating reactive systems by deduction. *ACM Trans. Softw. Eng. Methodol.*, 2(2):128–175, 1993.

[42] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.

[43] 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.

[44] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE : International Conference on Automated Software Engineering*, pages 263–272, 2005.

[45] Rajiv Gupta, David A. Berson, and Jesse Zhixi Fang. Path profile guided partial redundancy elimination using speculation. In *ICCL : International Conference on Computer Languages*, pages 230–239, 1998.

[46] 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.

[47] 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.

[48] 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.

[49] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171–181, 1988.

[56] 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.

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

[58] Mariam Kamkar and Patrik Krajina. Dynamic slicing of distributed programs. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 222, Washington, DC, USA, 1995. IEEE Computer Society.

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

[60] 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.

[61] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *ISSTA : Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 66–79, 1994.

[62] Jens Krinke. Static slicing of threaded programs. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 35–42, New York, NY, USA, 1998. ACM Press.

[63] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[64] 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.

[65] 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.

[66] Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2002. ACM Press.

[67] Ben Liblit, Alexander 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, 2003.

[68] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 289–299, 2003.

[69] Jin Lin, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, and Tin-Fook Ngai. A compiler framework for recovery code generation in general speculative optimizations. In *IEEE PACT : International Conference on Parallel Architectures and Compilation Techniques*, pages 17–28, 2004.

[70] 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.

[71] 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.

[72] 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*, 2005.

[73] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[74] 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.

[75] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

[76] Vijayanand Nagarajan, Dennis Jeffrey, Rajiv Gupta, and Neelam Gupta. Ontrac : A system for efficient online tracing for debugging. In *ICSM: International Conference on Software Maintenance*, 2007.

[77] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII : Proceedings of the 12th*

international conference on Architectural support for programming languages and operating systems, pages 229–240, 2006.

[78] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[79] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–31, New York, NY, USA, 2007. ACM Press.

[80] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07 : Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[81] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

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

[83] Peter G. Neumann. Risks to the public in computers and related systems. *SIGSOFT Softw. Eng. Notes*, 26(1):14–38, 2001.

[84] 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.

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

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

[87] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[88] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, pages 291–302, 2005.

[89] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP: ACM Symposium on Operating System Principles*, pages 235–248, 2005.

[90] 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.

[91] 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 : USENIX Symposium on Operating System Design and Implementation*, pages 303–316, 2004.

[92] Michiel Ronsse, Koenraad De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Commun. ACM*, 46(9):62–67, 2003.

[93] Michiel Ronsse, Koenraad De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *AADEBUG : Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, 2000.

[94] Radu Rugina and Martin C. Rinard. Pointer analysis for multithreaded programs. In *PLDI '99 : Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 77–90, 1999.

[95] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared memory applications. In *PLDI '96 : Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 258–266, 1996.

[96] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG : Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 69–76, 2005.

[97] Alexandru Salcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP : ACM Symposium on Principles and Practice of Parallel Programming*, pages 12–23, 2001.

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

[99] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.

[100] Sriraman Tallam, Rajiv Gupta, and Xiangyu Zhang. Extended whole program paths. In *IEEE PACT : International Conference on Parallel Architectures and Compilation Techniques*, pages 17–26, 2005.

[101] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 207–218, New York, NY, USA, 2007. ACM Press.

[102] 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.

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

[104] Larry D. Wittie. Debugging distributed c programs by real time replay. In *Workshop on Parallel and Distributed Debugging*, pages 57–67, 1988.

[105] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM Press.

[106] 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.

[107] 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.

[108] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[109] Min Xu, Rastislav Bodík, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA : Proceedings of the annual international symposium on Computer architecture*, pages 122–133, 2003.

[110] Min Xu, Mark D. Hill, and Rastislav Bodík. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS-XII : Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 49–60, 2006.

[111] 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.

[112] 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.

[113] Xiangyu Zhang. *Fault Location via Precise Dynamic Slicing.* PhD thesis, University of Arizona, 2006.

[114] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *PLDI '06 : Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 169–180, 2006.

[115] Xiangyu Zhang and Rajiv Gupta. Whole execution traces. In *MICRO : Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 105–116, 2004.

[116] 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.

[117] 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.

[118] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 81–91, New York, NY, USA, 2006. ACM Press.

[119] 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.

[120] Qin Zhao, Joon Edward Sim, Weng-Fai Wong, and Larry Rudolph. Dep: detailed execution profile. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 154–163, New York, NY, USA, 2006. ACM Press.

[121] 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.