A Framework for Recording & Replay of Software that Performs I/O

by

Rajiv Ramanasankaran

B.E. (Information Technology), Bharathidasan University, India, 2001

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

---

Dr. Jerry James, Chair

Dr. Douglas Niehaus, Member

Dr. Perry Alexander, Member

Date Thesis Accepted

Dedicated to my parents

Acknowledgments

First and foremost, I would like to thank Dr. Jerry James, my advisor and committee chair, who gave me the initial impetus to work on this thesis and helped me develop many ideas presented here. I cannot overstate the importance of his involvement in my graduate career. I also wish to thank him for giving me the opportunity to work with him as a Graduate Research Assistant. I am grateful to Dr. Douglas Niehaus, whose expert advice on any issue, was always indispensable. It was a pleasure working with him. I would like to thank Dr. Perry Alexander for being on my committee and reviewing my thesis.

I am grateful to my colleagues Sunil Penumarthy, Satyavathi Malladi and Rajukumar Girimaji for extending their help during the course of this thesis. The work done by them laid the foundation for this thesis. I wish to thank all the research assistants involved with the DSKI/DSUI project for helping me understand and utilize their framework.

I would like to thank the Altec team for their support and encouragement during the course of this thesis.

I wish to thank all my friends in KU, for being my family away from home during my graduate study. Last but not least, I am forever indebted to my parents for their understanding, endless patience and encouragement.

Abstract

Multithreaded solutions to problems of performance and computing are becoming more common. However, in many cases the temporal component of multithreaded programming is often difficult to understand and implement without errors. In a multithreaded program, independent tasks are mapped to threads executing concurrently. The inherently dynamic nature of events (context switches, signals and I/O) in such programs hampers consistent reproducibility of bugs. Using the Bthreads user-level thread library and the Clever Insight debugger it is possible to build and replay concurrency scenarios in a program, with respect to context switches and signals. This thesis extends the Bthreads/Clever Insight framework for I/O events by recording and replaying the contents of all I/O streams for completely reproducible execution.

The enhancements made to the BThreads library and Clever Insight debugger to support the new I/O recording/replaying framework are presented. The debugger GUI has also been modified to better synchronize the user interface with the GDB command line. Changes made to the debugger to automate the replay of program execution are also explained. The architecture implemented here enables users to develop more complex multithreaded applications by providing them with a powerful yet simple framework to reproduce executions.

Contents

List of Tables

List of Figures

List of Data Structures and Scripts

Chapter 1

Introduction

> "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
>
> Brian Kernighan

Taking into account the complexity and size of today's applications, such pessimism is not unfounded. Programmers need to be empowered with advanced tools which enable them to trace the execution of a program, including potentially complex system interactions.

Debugging is one of the most important and most common activities in program development. It can be considered as the process of removing bugs as well as understanding how the software really works. The debugger itself is a tool which serves this purpose. But classical debugging techniques such as breakpoints and single stepping cannot reproduce the temporal relations of concurrent computation components which were prevalent at the time the program was initially run. This important difference severely affects the assumptions programmers usually make while debugging and consequently makes bugs hard to find related to concurrency.

Other debugging techniques such as post-mortem trace debugging [5] can identify symptoms or failures, but not the actual bug itself.

Fixing bugs becomes a more pronounced problem in concurrent software due to the inherent dynamic nature of events like context switches, signals and input/output operations.

In this thesis, we present a framework to record the execution context of a program and then replay its execution while still allowing the programmer to use all the existing debugging techniques and tools. The importance of this framework lies in that each step of this framework is automated individually, thus automating the whole process and consequently taking a lot of tedium out of the debugging process. Using this framework, the programmer can specify how much control he wants at any instant. This framework also provides an option to synthesize new concurrency scenarios and thus puts an element of learning into the process by exposing the whole concurrency model to the interested student.

In the next chapter a brief survey of the various methodologies and tools that support replay of software and how our work relates to these is presented. Chapter 3 discusses the various modules which formed the basis of the recording and replaying framework. The interfaces provided by each module and the manner in which they interact is also discussed. Chapter 4 presents the implementation specific details of the design. The steps that must be taken to use the framework are laid out in Chapter 5. Chapter 6 discusses the testing of the framework, verifying reproducibilty and building of new concurrency scenarios. Conclusions and scope for future work are presented in Chapter 7.

Chapter 2

Related Work

Finding a generic way to debug and test any software is a hard problem. As evidence of this, almost twenty-five to fifty percent of the total system cost may be spent on these activities alone [11]. Programming methodologies to improve efficiency like concurrency introduces complexity increasing the cost further. It has been suggested that the best way to guarantee a bug free program is to ensure bugs are never introduced [10]. For these kinds of program verification techniques to work, the program behavior must be expressed as assertions on its input and output. But in many cases characterization of programs mathematically is an arduous task in itself. Further, constructing program paths and interleaving to cover all test cases is a cumbersome process. To this effect, programmers fall back on debugging techniques like user-controlled breakpoints, trace debugging, upon which almost all commonly used debuggers are based today. In this section we discuss the various other practices and tools used for debugging and testing of programs and outline how our work relates to them.

2.1 Deterministic Execution Testing

Deterministic testing uses a stored sequence of non-deterministic synchronization events and tests program behavior against these events by varying the inputs given to the

program. The sequence itself (containing non-deterministic synchronization events) is generated by performing source-level program transformations. "Tai & Carver" [9] discuss the language-based approach used to test and correct ADA programs. This method uses specific transformations for each of the different synchronization events in the program. In contrast, our method provides a standard interface for recording every event required to completely reproduce concurrent behavior without modifying the source of the program.

2.2 DejaVu

DejaVu [4] is a tool made by modifying the JVM (Java Virtual Machine) that captures the thread schedule information of non-deterministic Java programs and allows the programmer to replay these executions deterministically. The solution is independent of the operating system. The thread schedule information collected by DejaVu consists of the order of the synchronization operations and the shared variable accesses that took place in the program. This information is stored by DejaVu in terms of a global timestamp counter and is later used to reconstruct the thread schedules while replaying. This tool solves the problem of reproducing failures but doesn't take into account the input and output operations performed by the program that can change its course. Moreover, the changes made to the JVM for recording the thread schedule are specific to the synchronization mechanisms used by the program. In contrast, our framework provides the capability to replay the synchronization as well as I/O operations performed by the program and also allows the construction of new execution scenarios without being synchronization method specific.

2.3 JReplay: Instrumenting Java Bytecode

JReplay [1] is a tool for deterministic replay of multithreaded Java programs. Unlike DejaVu, JReplay doesn't depend on JVM modifications for replay. Instead, JReplay instruments the compiled class files by changing its bytecode. While replaying, JReplay overides the underlying JVM scheduler by transforming the compiled non-deterministic multithreaded program into a deterministic sequential program. It is able to accomplish this by ensuring that all threads except the current thread in the replay schedule are blocked. If the JVM scheduler has two threads to choose from, the replay fails. Thus it is mandatory for JReplay to handle all situations where threads are created,destroyed or in contexts where a transfer of control to another thread occurs naturally or is inserted according to an external schedule. Even though this method is innovative, it has some disadvantages. First, this framework depends totally on the format of the Java class bytecode. Secondly, JReplay doesn't support reproducibility of programs performing I/O operations. Moreover, specific transformations for each of the different synchronization events have to be re-written in case the API is deprecated.

Chapter 3

Design of the framework

3.1 Overview

Reproducing executions of concurrent software which has runtime constraints requires that the replay be performed in such a way that the execution context is reproduced fully. This means that we need a way to gather as much information from the software's first run to fully specify a particular execution scenario. One way in which this can be accomplished is to store the exact sequence of machine instructions executed along with the record of events which cause deviation in the flow of execution like signal delivery, context switching and I/O. However, such a specification is both cumbersome and unnecessary. A trace of basic block labels executed by the user code is sufficient information to specify the execution instead of storing the machine instructions. Insertion of context switch, signal delivery and I/O events at proper places in the basic block stream would then guarantee the exact and complete execution history while storing less information about program behavior.

After we have an event stream, we need a tool to deterministically replay the execution of the user program by generating signals, context switches and restoring data as defined by the history chronologically. One way to build such a framework without

changing the user program is to instrument the compiler as well as the libraries the user code uses and then use a replay tool to explicitly tell the program what path to take. The various tools utilized and constructed to build such a framework are discussed in the following sections.

3.2 Initial Pieces

3.2.1 GCC

"GCC" is an abbreviation for the GNU Compiler Collection which includes compilers for C, C++, Objective C and Fortran. The shorthand term formerly stood for the "GNU C Compiler" and is the name used when the emphasis is on compiling C programs. In this text, the mention of "GCC" refers to the C compiler alone.

Utilizing GCC's basic block profiling feature

One of our first goals is to capture the stream of basic blocks being executed by the program. GCC already provides an option to profile basic blocks. Based on this feature, a new compiler option is introduced to instrument the compiled code to produce a stream of basic blocks entered and executed as the program runs.

The instrumentation hooks inside the compiled code act as markers in the event stream and help us to reassert the execution path taken by the program during replay. One important aspect to consider and understand before proceeding is the difference between instrumenting GCC itself and adding instrumentation hooks to GCC. As we are interested in getting an event stream consisting of basic blocks being executed out of a running program, we have added a GCC option to instrument compiled code and not instrument GCC itself. In addition to having information about basic blocks, the interception and recording of context switch, signal delivery and I/O events still need to be performed in order to completely specify an execution history. In the following

section, we discuss BThreads, a user level multithreading library which provides us the capability of doing so.

3.2.2 BThreads

BThreads [8] is a POSIX compliant user-level multithreading library based on the Reactor [2], an object-oriented event demultiplexing framework.

In a user-level thread library, the whole process blocks if a single thread makes a blocking system call and as a result no other threads can be scheduled. To solve this problem, BThreads maps all threading primitives onto the Reactor framework (See Figure 3.1) , which assigns handlers for specific events and executes those events using a callback mechanism.

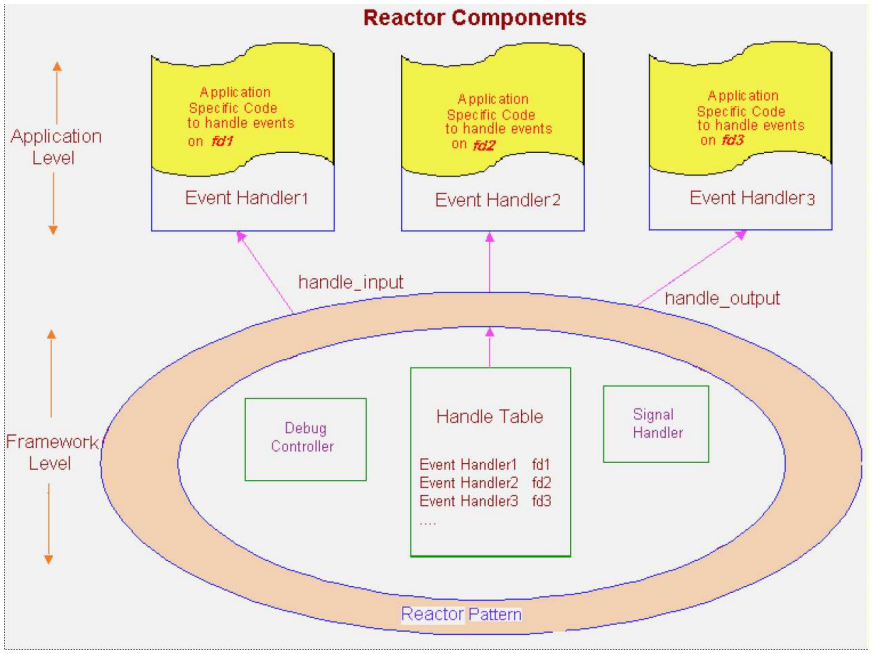

Figure 3.1: The Reactor Pattern for I/O handling ([2])

To handle events asynchronously, the Reactor framework offers an interface, an event handler, for the system calls that can potentially block. The handlers should

implement at least three methods. These methods are used to construct a callback mechanism whenever a particular event of interest can proceed without blocking. The methods are *handle_input*, *handle_output* and *handle_close*.

Reactor's role in the state of threads

The Reactor greatly influences the state of threads in the BThreads library. As mentioned earlier, I/O events register themselves and depend on the callback from the Reactor to proceed. As a result of the non-blocking characteristics offered by the Reactor framework, other threads are allowed to proceed. This is achieved by putting the thread (which made the blocking system call) into the wait state and then calling the scheduler to schedule another thread. Because the BThreads scheduler uses a preemptive scheduling policy, the state of the threads changes at least every Round Robin quantum.

BThreads' role in our framework

The BThreads library has the following features which can be useful in controlling and reproducing executions:

- The Reactor's serialized event handling mechanism allows BThreads to switch context between threads by transferring control through it.
- For every I/O system call, BThreads creates handlers which register themselves with the Reactor and get to proceed only when the operation can be performed without blocking. This is guaranteed by using the *select()* system call inside the Reactor which polls all file descriptors and allows only events that won't block. These blocking system calls are also encapsulated with wrappers to aid in the building of this framework.

- Currently, BThreads has instrumentation code which stores the receipt of context switch and signal delivery events in two separate text files. Having this information in text format limits the type of events and data we can store for replaying later.

BThreads provides wrappers around I/O system calls to route these events through the Reactor but doesn't provide any capability to replay them. A recording-replaying framework was build around the BThreads library to support reproducible executions of software which perform I/O operations.

We also need a tool to replay a recorded event stream. The widely used debugger, GDB (the GNU debugger), runs programs to be debugged as an inferior process under it. If we look closely, we find that GDB controls the inferior processes using the *ptrace* system call. This control can be used to specify what path a program should take while executing. This is a large piece of the requirements we have for a replay tool. Clever Insight [7] was built keeping these requirements in mind. We discuss Clever insight in the next subsection.

3.2.3 Clever Insight

Clever Insight is an enhanced version of GDB with support for BThreads. It uses the control mechanism provided by the BThreads library to deliver an event like a context switch or a signal to the application.

In order to debug a program linked against the BThreads library, the debugger must be conversant with the runtime structures and the various events happening inside the library. To achieve this, Clever Insight makes use of the thread debugger interface provided by the BThreads library. Clever Insight can identify threads, display their states, trace program execution and set thread specific breakpoints.

Clever Insight also provides a mechanism to replay context switches and signals

from the debugger but puts a lot of load on the programmer to accomplish this. Moreover, in the absence of a recording framework, the BThreads library creates two files, one with context switch events (*schedlogfile*) and the other with signal delivery events (*signalfile*). Later, if the user wishes to replay the execution scenario, he has to follow the following elaborate steps:

- First, manually gather the information from the *schedlogfile* or *signalfile* by interpreting the data stored in it.
- Then, explicitly use the GDB console to set the first conditional breakpoint at which a context switch or a signal event happened.
- After the breakpoint is hit, the user has to explicitly force a context switch or deliver a signal to replay these events. Since BThreads uses the SIGPROF signal as the timer expiry signal, the user has to deliver the SIGPROf signal in order to force a context switch.

This process is repeated for every event entry in the files. Also, since there is no timestamp information stored along with the events, it is very hard to construct the sequence in which the events took place from the two files.

This manual process hampers the testing of large concurrency scenarios involving many threads and events. This process was automated to make it easier for the programmer. Previously, Clever Insight didn't support the replaying of program that performs I/O operations. An interface was provided to Clever Insight through which it can direct the the BThreads library to restore and replay the I/O events too. Various alterations and additions to Clever Insight to provide both automated as well as controlled replay options to the user are discussed in detail in Section 4.6.5.

3.2.4 DSUI

The Data Stream User Interface (DSUI) is a tool developed at the University of Kansas to collect event traces and data from any user program or library. It supports a range of data collection options with different levels of detail and overhead. The use of DSUI in our recording framework enables us to specify and collect a series of timestamped events from both the BThreads library (context switches, signals and I/O events) and GCC (basic block trace). DSUI also comes with many postprocessing tools written in Python. A new filter has been specifically designed and added to simplify the number of steps to be taken by a programmer wishing to replay an execution scenario.

3.3 Design

The recording-replay framework has been designed to enable the user to specify an execution of interest and then exactly replay this execution using Clever Insight. During replay, the user can still use all of the existing features and techniques provided by the debugger. The user is not only allowed to automate the replay from start to end, but can also specify points of interest at which the execution should pause, and then either continue the replay or let the program take a different path of execution. We also ensure that the techniques applied by us to instrument the libraries and the instrumentation itself do not interfere with the program's execution or its replay. In the following sections we discuss how such a Recording and Replaying framework was built using GCC, BThreads, DSUI and Clever Insight.

3.3.1 Recording Framework

The recording framework is responsible for storing the execution trace of a program with enough information to later reproduce it fully. As we discussed earlier, this typically involves the basic block trace, the context switches and the invocation of system

calls that can change the behavior the program.

The recording process can be considered as a sequence of smaller events namely:

- interception of the event;
- bundling the information together in a record; and
- storing the record to a history.

Since the event stream might come from different source files or libraries, we also need to ensure the receipt and storage of the event in the proper order. DSUI comes to our rescue here. It provides a very easy interface to log events arriving from multiple sources and assures all events go to the same file in timestamp order.

DSUI allows us to classify events into familes and sub-events which can later be parsed into separate entities if needed. A record of an event is equivalent to an entity in DSUI. DSUI also provides us with the flexibility to tag an unlimited amount of extra data to the event record. But defining what constitutes a meaningful *event record* is also of prime importance. We take extensive advantage of DSUI's ability to tag extra data by creating our own data structures pertaining to specific events and storing them in the event stream.

3.3.1.1 Inserting instrumentation hooks into GCC

As discussed in Section 3.1 our first objective is to instrument the entry of each basic block and generate a stream needed for specifying the execution. GCC provides a feature by which we can get a list of basic blocks and the number of times each executed. This option (enabled by the *-ax* option) produces an output file *bb.out* which contains the basic block internal label, the basic block starting address and the function and source file it is in.

To access and eventually modify this information to suit our own needs, it is necessary for us to understand how GCC implements this option.

- First, each object file is compiled with a static array of counts, initially zero.
- In the executable code, every time a new basic block begins, an extra instruction is inserted to increment the corresponding count in the array.
- At compile time, a paired array is also constructed that records the starting address of each basic block.

Taken together, the two arrays record the starting address of every basic block, along with the number of times it is executed.

Based on the basic block profiling feature, we devised another feature to direct a stream of such *basic block events* to a file along with the basic block labels which were used by the compiler for accounting purposes. This stream of ordered events is used to ensure fidelity while replaying executions. However, we also need to insert events such as context switches, signal delivery and I/O into the stream as they occur. In the next subsection, the approach taken to accomplish that with BThreads (a user-level POSIX-compliant thread library) is described.

3.3.1.2 Instrumenting BThreads

BThreads already provides placeholders for inserting instrumentation to record relevant information, like CPU register state, the *currentthread id* and the stack pointer of the thread, at the time a context switch or signal delivery event happens.

Definition of a Reactor Event

One of the key elements missing in the BThreads library was the ability to record and replay I/O operations. Since the I/O events are demultiplexed with the help of the Reactor and the future state of threads in the library depends on the state of of blocking and non-blocking system calls at a particular instant, we record these Reactor events too in our global event stream.

One important point to note here is that the handlers inside the Reactor process an event only if they can do so without blocking on the system call. Therefore, we record the state of the handlers as well as *"no-operation"* events if the Reactor didn't send a callback to any of the handlers to make the actual system call.

Recording I/O streams

Whenever an I/O system call like *read* or *write* is made, the BThreads library routes the call to its corresponding wrapper function. Inside the wrapper, a check is performed to determine if the file descriptor is valid. If it is valid, an *iohandler* is created which contains the file descriptor, the thread ID and the number of times the handler is registered with the Reactor. This handler is then registered with the Reactor, the thread state is set to *waiting* and the scheduler is invoked. Since the system call is routed through the wrapper first, we access the I/O buffers of the call and record this information in our event stream (see Figure 3.2). The same principle can be extended to any system call and the event recorded. Finally, the arguments given to a program are also recorded and replayed since they too impose constraints and affect the execution of the program. This is done by adding a call to a routine in the *main* function of the program with *argc* and *argv* as the arguments.

The need for postprocessing the recorded event stream

Since the data that needs to be recorded varies from event to event, we use different data structures for different events. We later tag these data structures to the event by passing it to the optional *extra data* argument in the DSUI logging function. The different types of events recorded are controlled with a DSUI configuration file.

After the program executes, a file name having the pattern *bert-events.pid* is created in the temporary directory of the file system for each program execution. A single stream of events can be used to ensure fidelity while replaying a particular execution.

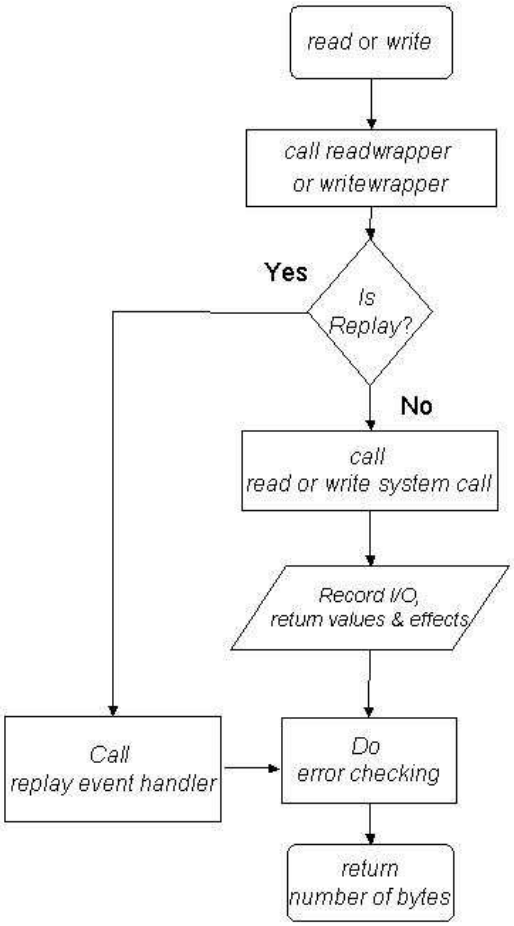

Figure 3.2: Read/Write system call wrapper

But the process of accessing different families of events out of a single event stream, especially when querying events from different parts of the recording-replay framework would require a much more elaborate approach. For this reason, a special DSUI filter is constructed to break this event stream into separate families.

3.3.2 Postprocessing using DSUI filters

We note in the subsequent sections that events like context switches and signals are played back using the Clever Insight debugger and I/O events are replayed using the BThread library wrappers. As the mechanism and the choice of the event restoration point vary considerably during replay, we need an intermediary postprocessing stage to separate events by family. To simplify the number of steps involved, we also generate a configuration file. This configuration file contains the paths to the various files generated and is used as the single input given to Clever Insight to replay the program. At the end of this process, we have:

1. a configuration file;
2. a data stream containing signals and context switch events;
3. a data stream containing I/O events;
4. a data stream containing Reactor events; and
5. a data stream containing the arguments supplied to the program during its first run.

3.3.3 Replaying framework

The design of the replay framework is influenced greatly by the design of the recording framework. This is because we are re-creating the execution context of a program during replay. This lends us a unique advantage of substituting replay code in the

place of the recording code. As mentioned earlier, a majority of the events take place while inside the BThreads library. Even though this substitution can only be useful for events that take place due to the linear execution of library code (like I/O events) we realize this approach is simple. As for other events like context switches and signal delivery, which are only to be replayed when the program reaches the correct place in its replayed execution , we replay them by enhancing the Clever Insight debugger.

We now discuss the structure of the replaying framework which is used to perform incremental replay of a user program from a recorded event stream.

3.3.3.1 Replaying events from BThreads

For reproducing context switches and signal delivery events, Clever Insight makes use of the ability of the debugger to set breakpoints and deliver signals. For example, replaying a single context switch event involves setting a conditional breakpoint at a particular program counter value. The breakpoint is always present at that address but it is ignored until proper context is reached during replay of the n^{th} loop iteration of the basic block.

If we use a similar approach to reproduce I/O events, we must alter the debugger by setting breakpoints at system calls and provide functions which basically the stored I/O streams and copy them to the inferior process's data structures. The same applies for events coming from the Reactor. If we examine how system calls really work, we find a better solution.

Programs execute system calls by trapping into the kernel. The kernel executes the code on behalf of the user process and may write to the user address space. For replay to work we must be able to access and modify the data in the range of addresses written to or read from.

Determining the range of addresses is simple. For example, the arguments to the *read* system call specify the address of the buffer where data is to be written and the

length of this buffer. The return value of these system calls specify the number of bytes read or written. But where would we get access to these addresses while replaying? The solution is pretty straightforward. We take advantage of the fact that BThreads already provides wrappers for the basic I/O system calls in order to route them through the Reactor. We use a flag in the library to specify whether we are in *recording mode* or *replay mode*. Whenever an I/O system call is performed, the library intercepts the call and directs it to our wrapper functions. Under the *recording mode*, we construct and store an event record into the global event stream using DSUI (see Section 3.3.1). But while in *replay mode* we use a set of internal *replay event handlers* to:

- Copy the recorded streams to the I/O streams of the program;
- Restore the return values of the system calls from the recorded stream;
- Restore the value of the *errno* variable depending on the semantics of the system call;
- Ensure the correct future state of all threads by replaying Reactor events, i.e. by restoring the state of the event handlers at that instant; and
- Restore the command line arguments given to a program with the stored arguments from the recorded event stream.

Advantages of our approach

- This method avoids the unnecessary setting of numerous breakpoints at system calls and provides the flexibility of completely switching off the replaying framework by changing a single flag.
- This method also offers the user the unique opportunity to replay an execution till some point of interest and let the program take a different execution path thereafter.

- This method also allows the user to construct a specific interleaving by toggling the mode between *replay* and *normal* at any point of the execution.

3.3.3.2 An Enhanced Clever Insight

Conventional debuggers work on the principle of setting breakpoints at specific places in the code and allowing the programmer to trace the program execution. Many times a bug in the code is dependent on the computation context, including time. Reproducing the execution context would enable the programmer to perform incremental as well as controlled debugging and is one of the prime goals of the replaying framework. The integral and perhaps most important part of this framework is the replaying tool: Clever Insight.

The enhanced Clever Insight has been built to reproduce the execution context and hence puts the control of the replay machine into the hands of the user. For this purpose, the enhanced Clever Insight provides the user with two basic control modules which are used collectively for the replay functionality. They are :

- the context switching & signals replay module; and
- the I/O events replay module.

Context switches & signals replay module

This module deals with the automatic setting up of the next context switch or signal delivery event specified in the recorded event stream. Doing this in an incremental manner helps the programmer to stop the replay at a certain point and then either construct his own execution scenario or use traditional debugging methods. It also enables him to stop in the middle to change the state or value of any variable and notice the effects if the replay is continued.

I/O events replay module

This module basically exposes a control routine inside the BThreads library using which the user can choose to replay I/O events. The actual functionality is also embedded in the BThreads library itself (See Section 3.3.3.1).

Using these modules, the programmer has full control over the replayed program's behavior.

3.3.4 Meeting Design Goals

Traditional debugging techniques work well for serial programs that do not interact with any other dynamic entities. However concurrency and input/output actions during execution alter their behaviour and results. Another technique, tracing , gives a better perspective of how the program behaved in the past but either involves the arduous task of inserting print statements or doesn't expose the concurrency model of the program to the user. We started out with the idea of removing these hurdles in the way of the programmer. The design discussed in this thesis coupled with the control the BThreads library provides removes these hurdles in the following ways:

Non-intrusive framework: The recording & replaying framework doesn't require changing the program's source code as other tracing techniques do.

Interface to the concurrency model: The framework exposes the concurrency model to the programmer and gives him full control over the program he is replaying.

Supports traditional debugging primitives: Since the execution context is restored during replay, the programmer is able to reproduce the exact interleaving of events without losing the ability to use traditional debugging primitives.

Creating new scenarios: Our design helps the user to construct and learn about the effects of new scenarios with total or partial reproduction of the environment each time.

Automation and control: This framework introduces a method of automated or controlled incremental replay by providing a new level of interactivity in the process of debugging.

We discuss the implementation of the design in the next section.

Chapter 4

Implementation details

We now discuss how we implemented this framework.

4.1 Specifying events and families in DSUI

The Data Stream User Interface (DSUI) allows us to categorize events as belonging to different families. Table 4.1 shows the various families we defined and the corresponding events they have.

This categorization helps us to filter out specific event entities of interest from the recorded event stream. The name of the file used to specify these families and events is *namespace.dsui*. DSUI comes along with a utility *dsui-parse* which takes the *namespace.dsui* file as input and generates the necessary header files having declarations of

DSUI Family	*Events in the family*
GCC	EVENT_BB_ENTER
MAIN_ARGS	EVENT_APP_ARGS
BTHREAD_LIB	EVENT_SCHED
IO_WRAP	EVENT_IO_OPEN,EVENT_IO_CLOSE EVENT_IO_READ,EVENT_IO_WRITE
SIGNAL	EVENT_SIGNAL
REACTOR_WRAP	EVENT_FD_SET

Table 4.1: DSUI families and events

logging functions that we eventually use to instrument events. It also assigns unique IDs to events automatically. The DSUI library itself is initialized by calling DSUI_INIT(*prefix*, *enable file*). The *prefix* here is the one which should be added to the final DSUI output file, and the enable file contains the list of events that should be enabled or recorded.

4.1.1 Recording data with DSUI

The syntax of a DSUI logging function is

*DSUI_EVENT_LOG (family_name, event_type, tag, extra_data_length, (void *)extra_data)*

Since we have to store specific structures and binary data collectively to specify an event, we bundle the information together in our own data structures. This record is then stored to the binary stream by utilizing the optional *extra_data* parameter of the DSUI_EVENT_LOG function. The content of the *extra_data* and the manner in which it is stored in the event stream is discussed in the implementation sections of the various events.

4.2 Recording Basic Block Events

As discussed in section 3.2.1, we make use of the basic block profiling code inside GCC for developing our own method of recording a stream of basic block events.

How does basic block profiling work?

GCC maintains an internal array of basic block structures which it uses for keeping an account of basic block entry events. Normally profiling is switched on by passing the *-ax* option to the compiler. This option sets the value of an internal flag that causes the compiler to put in calls to specific routines in the library *(libgcc2.a)* at the entry of every basic block. Every time a program executes one of the basic blocks, the internal

array element representing the accounting information of that particular basic block is updated. This updated information is the starting address of the basic block and the number of times it has been executed. The cumulative data in the array is finally recorded in a file *bb.out* when the program exits.

Recording a stream of basic blocks

To produce a stream of basic block events, we introduce a new option, *-bert*. The new option forces the compiler to output code that executes our own special routine in the *libgcc2.a* library. This routine packages the relevant information into a data structure as shown in Listing B.2. We record this structure by passing it as the *extra_data* argument in the call to DSUI_EVENT_LOG.

Listing 4.1 Basic Block event structure

```
/*To be stored as extra data in an event*/
typedef struct bb_enter_struct {
    unsigned long block_start_address;/*The basic block start addr*/
    unsigned long bb_blockno;/* The basic block label*/
} bb_enter_t;
```

The address of the current basic block, and the number of times this particular basic block had been executed are stored in an array *_thread_handles[currentthread]* inside BThreads. During replay, this information helps us to identify the exact conditions under which a thread should switch context or receive the recorded signal.

4.3 Recording I/O Events

I/O events can be broadly classified into:

- Reactor events: events involving recording the list of active handlers and their states;

- System call events: events involving recording of buffers and effects of system calls; and
- Command line arguments: events that record the command line arguments given to a program.

4.3.1 Recording the list of handlers and their states in the Reactor

The BThreads implementation of the Reactor supports three basic callback routines for the handlers that register with it. These are:

- handle_input: handler called whenever data is ready to be received;
- handle_output: handler called when data is ready to be written to a buffer; and
- handle_close: handler used to check whether an event has unregistered itself from the Reactor.

The Reactor polls the set of file descriptors belonging to the I/O events (using the *select* system call) and calls the appropriate handler to handle that event. The Reactor performs these operations in the *handleEvents* method. Since in the *handleEvents* method, the Reactor can handle events for zero or more handlers, we construct a list of handlers processed with their states and store this information in the event stream. The data structure used to accomplish this (Listing 4.2) contains the index of the handler and the event type .i.e input or output. If none of the handlers receives a callback from the Reactor, we still consider it as an event and record a *no-operation* event by storing NULL values.

We finally bundle this list in the buffer and pass it as the *extra_data* parameter in the call to DSUI_EVENT_LOG.

Listing 4.2 Reactor event structure

```
/* To be stored as extra data in an event*/
 typedef struct fdstruct{
      int handlerListIndex;//the handler index
      char input_or_output;//'o' means output and 'i' means input
      struct fdstruct *next;
  }fd_event_t;
```

4.3.2 Recording system calls

The recording of system calls involves:

1. Recording the buffers passed to the system calls; (e.g., in the *read* system call: *read(int fd, void *buf, size_t count)* we record the content of the parameter *buf*).

2. Recording the bytes read or written; (e.g., return values of the *read* and *write* system calls).

3. Recording of the effects of the system call after it is performed. (e.g., the value of *errno*)

4. Recording of file descriptor values returned by the Operating system (e.g., in case of the *open* system call.)

Since the syntax and arguments of the *read* and *write* system calls are almost the same, we use the same structure (Listing 4.3) to record these events.

Listing 4.3 Event structure for *read/write*

```
/*To be stored as extra data in an event*/
typedef struct io_rw_struct {
    unsigned long int currentthread_id;/*the currentthread id */
    int fd_no; /* the file descriptor */
    int io_data_len; /*length of the buffer*/
    int errno;
} io_rw_t;
```

These structures are filled with the data from the system call buffers. The return values and their effects (*errno*) are also stored. This structure is then passed as the *extra_data* argument in the call to DSUI_EVENT_LOG.

In the case of the *open* and *close* system calls we use the structure shown in Listing 4.4. For these system calls, it is enough for us to store the file descriptor values and the effects of making the call like *errno* and the return value. This structure is also passed as the *extra_data* argument in the call to DSUI_EVENT_LOG.

Listing 4.4 Event structure for *open/close*

```
/*To be stored as extra data in an event*/
typedef struct io_oc_struct {
    unsigned long int currentthread_id;  /* the currentthread id */
    int fd_no; /* the file descriptor */
    int retval; /*Return values open/close calls*/
    int errno;
} io_oc_t;
```

4.3.3 Recording command line arguments

For recording the command line arguments, the user has to call the routine

INIT_MAIN_ARGS(&argc,&argv)

from the *main* function. The routine should be called with the addresses of the arguments to main. This solves a two-fold purpose:

- While recording, the arguments to *main* are accessed and stored in the event stream from these parameters; and
- While replaying, the same function is used to restore the arguments from the recorded event stream.

Each argument is stored as a separate event in the event stream as the *extra_data* of the event EVENT_APP_ARGS since they need to be restored as members of an array in exactly the same sequence as specified by the user initially.

4.4 Recording context switch and signal delivery events

Context switch events:

The BThreads library uses the *itimer_prof* interval timer for generating the scheduling timer. Whenever the timer expires, the SIGPROF signal is delivered and the registered signal handler is invoked.

Initially, the *sigaction* call was specified to call the signal handler with only one argument containing the signal number that was delivered. This was done by passing the handler function to *sa_handler*. Since we need to store the instruction pointer at the moment the signal was delivered, we require access to the context the thread was executing in. The system call *sigaction* allows to specify a signal catching function which will be entered along with two additional arguments, namely the *siginfo_t* structure (which contains the explanation of the reason why the signal was generated) and a pointer to an object of type *ucontext_t* (refers to the receiving process' context that was interrupted).

We get the value of the instruction pointer by accessing the *uc_mcontext.gregs[EIP]* member of the *ucontext_t* structure. The basic block number and the basic block counter values are accessed from the structure (*_thread_handles[]* array) that BThreads already provides. The thread id of the currently executing thread when the SIGPROF was generated is accessed from the variable *currentthread*. This information is then collectively stored into a *sched_t* structure (Listing 4.5) and then stored as the *extra_data* argument in the call to DSUI_EVENT_LOG.

Signal delivery events:

A signal delivery event is specified by the *sgnl_t* structure shown above in Listing 4.6. The information needed to record this event is fetched from the same data structures as the ones used for the context switch events. Finally, this event is also stored in

Listing 4.5 Event structure for a context switch

```
/*To be stored as extra data in an event*/
typedef struct sched_struct {
    unsigned long int contextswitch_thread_id;   /* the thread_id */
    unsigned long eip_address;/* Program counter value*/
    unsigned long bb_blockno; /* basic block number/label*/
    unsigned int bb_counter;/* basic block counter */
    unsigned int contextswitches;/* number of context switches*/
} sched_t;
```

the event stream as the *extra_data* argument in the call to DSUI_EVENT_LOG.

4.5 DSUI filter for BERT

To ease the number of steps taken by the programmer to use our framework, we designed a special filter by utilizing the existing family filters available with DSUI.

This DSUI filter (enabled by passing the *-bert flag*) takes in the recorded event stream (bert-events.pid) and arranges the events into the following files:

1. bert-event.pid.io: Events of the I/O family like IO_READ, IO_WRITE, etc.

2. bert-event.pid.reactor: Events belonging to the REACTOR_WRAP family.

3. bert-event.pid.schedsgnl: Events EVENT_SIGNAL and EVENT_SCHED.

4. bert-event.pid.mainargs: Events representing the command line arguments given

Listing 4.6 Event structure for signal delivery

```
/*To be stored as extra data in an event*/
typedef struct sgnl_struct {
    unsigned long int currentthread_id;/* the thread_id */
    unsigned long eip_address;/* Program counter value*/
    unsigned long bb_blockno;/* basic block number/label*/
    unsigned int bb_counter;/* basic block counter */
    unsigned int contextswitches;/* number of context switches*/
} sgnl_t;
```

to a program.

5. bert-event.pid.config: A configuration file which acts as the single input to Clever Insight to reproduce the execution. This file contains the commands for GDB to set up the replay framework and also provides the path of all the other files.

A sample configuration script is shown in Listing 4.7. We will discuss this script's purpose and the commands in the next section.

Listing 4.7 A sample bert-events configuration file produced by the DSUI filter

```
delete breakpoints
handle SIGPROF nopass
break main
commands
call setMode(1)
activate_replay
replay_schedsgnl_file /tmp/bert-events.565.schedsgnl
call setReplayIOStreamBinaryFile ("/tmp/bert-events.565.io")
call setReplayFDSETStreamBinaryFile ("/tmp/bert-events.565.reactor")
call setReplayMainArgsBinaryFile ("/tmp/bert-events.565.mainargs")
end
run
```

4.6 Implementation of the Replaying framework

4.6.1 Specifying the Replay configuration file to GDB

As mentioned in Section 4.5 we just need to supply a single configuration file to Clever Insight to set up the replaying framework. The commands in the configuration file are executed line by line by using the *source configfile* command in GDB. The commands in the file (Listing 4.7) and their purpose are discussed below:

1. *handle SIGPROF nopass*: This command is executed before the replay process since context switches in the BThreads library are triggered by a SIGPROF signal,

which is sent on expiration of a timer. For this reason, the SIGPROF signal should not be delivered to the program and hence this command.

2. *break main*: This command sets a breakpoint at the *main* function of a program.

3. *commands*: The *commands* command in GDB allows us to specify a set of commands to be executed at a breakpoint. If given without an argument the breakpoint is the last one set. The following commands are executed when the breakpoint at *main* is reached.

4. *call setMode(mode_flag)*: This is a utility function in BThreads which toggles the library between *recording mode* and *replay mode*. Setting a value of 0 (or *recording_mode*) tells BThreads to execute the code for recording events. A value of 1 (or *replay_mode*) changes the mode to *replay*.

5. *activate_replay*: This new command has been added to GDB and serves two purposes:

   - It sets the replay breakpoint for the next context switch or signal delivery events from the recorded event stream.
   - It also serves as a flag to distinguish between breakpoints set by the user and breakpoints set by the replaying framework. This difference is particularly useful in cases where the user wishes to use traditional debugging primitives (like single stepping/breakpoints) during the replay. The *activate_replay* flag is automatically disabled after replaying an event to ensure the setting of this flag only from our script.

6. *replay_schedsgnl_file*: This command has been added to GDB to specify the location of the file containing context switch and signal delivery events (generated by the DSUI filter). It feeds the file to the internal *Context switch and Signal replay module* discussed in Section 3.3.3.2.

7. *call setReplayIOStreamBinaryFile, setReplayFDSETStreamBinaryFile, setReplayMainArgs-BinaryFile*: These calls are utility functions which have been provided as part of the replay event handler framework described in Section 4.6.2. The commands make the replay event handlers conversant of the paths to the files containing the various events. The replay event handlers query the files incrementally and replay the events.

8. *end*: Marks the end of the *commands* command.

9. *run*: Executes the program under GDB control. Note that we don't have to specify any arguments to the *run* command to replay the program since the replay event handlers automatically restore the program arguments when the replay starts.

4.6.2 Replay Event handlers for I/O events

A replay event handler is the function responsible for reading the data from the recorded event stream and restoring them appropriately to replay the event. Every event that needs to be replayed has its own replay event handler. In the following sections, we discuss how these are vital to the I/O replay framework.

4.6.3 Implementation of the I/O events replay module

The I/O events replay module was discussed in Section 3.3.3.2. Here we discuss its implementation.

As mentioned in the design, the same wrappers used to record I/O events are used to replay them. Therefore, the BThreads library works in either the *recording_mode* or *replay_mode*. This is implemented by changing the value of the variable *mode_flag*. The utility function through which we can control the mode inside the BThreads library is *setMode(mode)*. It takes a single argument which can be 0 (*recording_mode*) or 1 (*replay_mode*).

Under *replay_mode*, instead of making the actual system call, we invoke the event's specific replay event handler. For example, the replay event handler for the *read* system call restores the buffer and the number of bytes read. We also restore the value of the *errno* variable from the event stream. Other system calls are also replayed in the same fashion.

4.6.4 Implementation of the Reactor Replay Module

As discussed in Section 3.3.3.1, we would need to restore the state of all registered handlers every time the Reactor calls the handleEvents method. This might also include *no-operation* events. The normal function of the *handleEvents* function is to poll all the file descriptors using the *select* system call and callback the appropriate handler of that file descriptor on which the operation can be performed without blocking. While replaying we do not execute the *select* system call at all. Since we record the list of handlers that were processed by the Reactor, in order to replay the event it is enough for us to process the callback functions of the same set of handlers. Since we bundled a collection of such handlers together inside the *extra_data* part of the event, the Replay event handler for the Reactor constructs a linked list of handlers and their type from the recorded event stream. When this list is returned to the *handleEvents* function, we traverse the list and process the handlers in succession. Since we replay the exact set of handlers, the states of the threads are restored correctly.

4.6.5 Implementation of the Context switch and Signal replay module

In Section 3.3.3.2, we mentioned the *Context switch and signal replay module* that has been implemented inside Clever Insight. The context switch and signal replay module has been developed as an add-on feature for Clever Insight. We have been careful not to affect the functionality of any other debugging features.

In order to replay a context switch, we have to know the exact conditions under which the context switch happened. We gather this information from the recorded event record, comprised of:

- the thread id
- the program counter
- the basic block number the thread was executing
- the basic block counter

After the first context switch event occurs during replay, we have to set the next one automatically. We can use the *commands* command of GDB to specify the commands to be executed when GDB hits a breakpoint. We would have to use the *commands* command again in the next replay breakpoint we set to continue the replay. Unfortunately, GDB doesn't allow nesting of breakpoints that specify commands to be executed when they are hit.

Nesting breakpoints: The problem and the solution

The breakpoint handling code in GDB doesn't support the nesting of breakpoints which have a set of commands attached to them. It simply ignores the nested commands.

Listing 4.8 Nested breakpoints in GDB

```
break <address> <condition>
commands
break <address> <condition>
commands
.................
.................
end
end
```

For example, if the user sets a breakpoint like the one shown in Listing 4.8, the commands of the nested breakpoint are not executed at all. The reason for this is that GDB

reuses the same data structures to store the breakpoint commands of the first breakpoint. But in order to set the next replay breakpoint, we need to make this possible. This problem was solved by modifying the breakpoint handling code in such a way that it deals with the replay breakpoints we set differently. This is accomplished as follows: After all the commands of the first breakpoint have been executed, we check whether one of the commands executed was *activate_replay*. If so, then the present breakpoint that was hit is a replay breakpoint. After making sure that it is a replay breakpoint, we safely set the next breakpoint by calling the *setNextBreakpoint* function inside the *Context switch and Signal replay module*. This function is the one which queries the event stream and sets the next breakpoint by the process explained in the following section.

Script for setting the next replay breakpoint

This module is responsible for setting the next replay breakpoint incrementally. To achieve this functionality, we first query the next context switch or signal event from the event stream and write a temporary script file in the */tmp* directory. This file is created with the name pattern *bert-events.pid.schedsgl.schedtmp*. A sample script file is shown in Listing 4.9.

Listing 4.9 A sample script generated by the Context switch and Signals replay module

```
break *0x40222654 if _thread_handles[1].block == 29 \
&& _thread_handles[1].count == 48
commands
disable_last_breakpoint_hit
activate_replay
signal2give  27
end
```

The *source* command is then used to execute the commands in the script file.

1. *break* < `condition` >: In this example script, this command sets a conditional breakpoint at the address 0x40222654 where thread number 1 was executing the

29^{th} basic block for the 48^{th} time. These conditions are taken from our recorded event stream and formalized into a breakpoint.

2. *commands*: The *commands* command in GDB allows us to specify a set of commands to be executed at a breakpoint. If given without an argument the breakpoint is the last one set. The following are the commands executed when the breakpoint at main is hit.

3. *disable_last_breakpoint_hit*: This command was added to Clever Insight. It is used to disable the last breakpoint that was hit. This is essential because otherwise the program may hit the same breakpoint again and stop after the SIGPROF signal is delivered to it. This hampers automated replay and hence the command is used to solve that problem.

4. *activate_replay*: The use of this command is explained in Section 4.5.

5. *signal2give*: Whether it is a context switch event or a signal event, eventually a signal is delivered to the program. This command tells the replay module which type of signal to give the program after the breakpoint is hit. For example, the signal number 27 is SIGPROF which means we are replaying a context switch event.

6. *end*: Marks the end of the *commands* command.

After the above script is executed, the decision process differs considerably for *automated replay* and *controlled replay*.

Automated replay

If the user specified automated replay by using the command *automate_replay* at the GDB prompt (e.g., for replaying test suites), we take the following steps in the *Context switch and signal replay module* when any of the replay breakpoints are hit:

1. Set the next replay breakpoint from the event stream.

2. Give the inferior the recorded signal (SIGPROF or other) automatically.

In this manner, (until the program is explicitly stopped by a SIGINT signal or till the program exits), the program continues to replay using the events from the recorded event stream.

Controlled replay

If the user specified controlled replay using the command *controlled_replay*, we just set the next replay breakpoint. When the breakpoint is hit, control goes back to the user. At this point, the user is free to either:

- give the program a SIGPROF signal (which forces a context switch) and tell it to continue; or
- switch back automatic replay by using the *automate_replay* command and then continue; or
- use traditional debugging techniques and use the *handle SIGPROF pass* command to let the program take its own execution path (this includes changing values in memory and other data and then letting the program to continue); or
- construct a concurrency scenario by using the commands Clever Insight provides, like *switch_to_thread(thread_no)*.

Chapter 5

Recording & Replaying executions with the framework

In the earlier chapters, we discussed the ability of the Recording-Replay framework to record executions and then replay them by making use of the extensions provided inside BThreads and Clever Insight. Here, we demonstrate the procedure the user must take to accomplish the same. We also show how we can control the replay and even construct new scenarios with the enhanced Clever Insight.

5.1 Recording program execution

Before the program is executed, there are a few steps the user must take to ensure proper working of the framework:

5.1.1 Prerequisite setup of the framework

- To ensure a stream of basic block events from the running program, the *-bert* flag must be supplied to *gcc* when compiling.
- To take advantage of the facilities offered by the framework, the program to be

replayed should be linked against the BThreads library. The user should also pass the *–wrap* option to the linker so that all I/O calls to the system functions like *read*, *write*, etc., are resolved to the wrappers provided inside the BThreads library.

- To enable recording of events into a event stream using DSUI, a file named advanced-enabled.dsui should be present in the directory from where the program is invoked. The structure and content of the file is listed in Appendix B.

After this prerequisite setup, the program is ready to be run. After the program has exited, a file with the pattern bert-events.pid is created in the */tmp* directory of the machine. This file is the output generated by DSUI and contains all the events (context switches, signals and I/O) required by the replaying framework.

5.1.2 Intermediary step using DSUI

An intermediary step is needed to parse and separate events according to their DSUI families. This is accomplished using the DSUI filter with the *-f bert* flag. The syntax of the command to process the DSUI file is
filter -f bert namespacefilepaths -rb -wb /tmp/bert-events.pid
This process produces a configuration file with the pattern bert-events.pid.config and several other files containing specific families of events in the */tmp* directory. The bert-events.pid.config file is the only file needed by the replaying framework.

5.2 Using the enhanced Clever Insight for replay

The bert-events.pid.config file contains all the nexessary information for Clever Insight to replay the execution of the program. The steps to be followed are outlined below.

1. First, we load the program in Clever Insight. This can be done by either giving

the executable name as one of the arguments to Clever Insight at the command line or the user can start up the GUI and select the executable from the *File* menu. The user can also specify the path using the *file* command at the Clever Insight prompt.
(gdb) file < executable >

2. After the file is loaded in Clever Insight, the user should use the *source* command and specify the path to the *bert-events.pid.config* file.
(gdb) source bert-events.pid.config
At the end of this step, Clever Insight is ready to replay the execution of the program.

3. Now the user has the option of deciding whether he wants the replay in *Controlled replay mode* or *Automated replay mode* .

Controlled replay

Controlled replay mode is switched on by default. Under this mode, after a replay breakpoint is hit, control comes back to the user. After each replay breakpoint is hit, the user must explicitly deliver the SIGPROF signal by executing the *signal SIGPROF* command at the Clever Insight prompt.
(gdb) signal SIGPROF

Automated replay

If the user wishes to do the replay in *Automated replay mode*, he must pass the *automate_replay* command at the Clever Insight prompt. The program executes till it exits or the user interrupts it (by issuing the SIGINT signal).
(gdb) automate_replay

The mode can be changed back to *Controlled replay* anytime during the replay by passing the *controlled_replay* command.

4. Under both modes, the user can still use any of the traditional debugging techniques like setting breakpoints, analyzing and modifying memory, etc. Clever Insight already allows the user to do state exploration of BThreads. Since I/O streams are also restored, the user can query and modify them too.

Chapter 6

Evaluation

The framework that has been built can be broadly classified into smaller components: the recording framework and the replaying framework. In this section, we discuss the various testing measures that were employed to test the working of the above two components and the framework as a whole.

6.1 Event stream comparison under a special mode

The event stream emitted by the recording framework contains important information required by the replaying framework for faithful reproduction of the execution. A special mode was integrated with the replaying framework which checks the contents of the recorded event stream and the event stream emitted by the program while replaying. Both outputs are compared using the *diff* tool. This option enabled us to prove that there was no discrepancy between the recording and replaying event streams of the program. A difference in the trace of execution would have meant the program took different paths while recording and replaying.

6.2 Testing I/O reproducibility

Checking the contents of all I/O streams at every stage of program execution while recording and replaying can test reproducibility of I/O operations. If we are able to reproduce the I/O streams completely from the recorded stream, we gain confidence that the I/O was recorded and replayed successfully.

For this test, a program to copy the contents of one file to another was written. The copying is done from the source to the destination file using standard system calls like write and read. The contents of the I/O streams are stored in the event stream along with other events like context switches and signals. While replaying, the calls to read the contents of the source file are redirected to the replay event handler function wrappers, which populate the buffers with the data from the recorded event stream. The other system calls for opening, writing to and closing the destination file were not wrapped, allowing the destination file to be re-created on the disk from the recorded event stream. After replay, the contents of the original source file and the re-created destination file are compared. Complete reproduction of I/O streams is then guaranteed by using the *diff* utility on these files and seeing whether they differ. The same test is also performed by using a multi-threaded version of the copy program that makes multiple copies of the same file on disk using multiple threads.

6.3 Testing Program flow

While reproducing the execution of programs, it is also very important to test the faithful reproduction of thread states and interleaving. For testing this feature, the dining philosophers problem was executed and a specific interleaving that led to a deadlock is noted. When the execution was replayed, the interleaving was compared to the recorded interleaving. We also note that the interleaving led to a deadlock with repro-

duction of thread states.

6.4 Creating a specific interleaving

One of the features supported by this framework involves the creation of a specific interleaving of events. For testing this feature, we used the execution history of a specific case of the dining philosophers problem that led to a deadlock. We reproduced the execution of the history till a point of interest but before deadlock occurred. At this point, we changed the thread states using Clever Insight in such a way that the deadlock is avoided or deferred. During this test we also record the event stream emitted bythe replaying of the program. This newly constructed event stream was then fed into our replaying framework to confirm and reproduce the exact interleaving that we set up. A test was also performed by creating a deadlock scenario and replaying it using the framework.

Chapter 7

Conclusions and Future Work

Traditional debugging methods provide little insight into the cause of abnormal application behavior. Software may be long-running and have complex interactions with the environment. Reproducing bugs by re-executing a given program may be impractical due to changes in the program environment.

In this thesis, we have presented the details and implementation of an extensible framework for recording and replaying executions of programs.

- We have investigated the cause of deviation in program behavior during its multiple runs and formalized a process of recording it in a history and replaying it later. We have demonstrated this point by replaying contents of input/output streams along with context switches and signals.

- The framework can be used to easily track program bug symptoms back to their causes.

- The whole process has been automated to reduce the number of steps needed by the user of the framework. We have also designed and built a replay engine into Clever Insight for incrementally replaying executions in controlled as well as automated fashions.

- The framework also provides the user the ability to analyze and modify application state and data during replay while still allowing the use of traditional debugging primitives. This flexibility empowers the user to interactively define what path an application should take after being replayed for some time, which would be impossible or very difficult to do using conventional debuggers.

The following are further enhancements that could be integrated later with this framework:

- The concept of reproducible executions can be extended to distributed systems by simulating the underlying network on top of BERT.
- The framework can be used to reproduce executions of Java programs by porting the JVM on to BERT.
- A richer set of GUI functions can be integrated with Clever Insight to make debugging and creation of specific thread interleaving even more easier.
- The set of system calls that could be replayed can be extended to cover larger program types.
- After porting User-mode linux on to BERT, this framework can be used to replay executions of the operating system.
- The event stream history tends to get larger as the program complexity and functions increase. The stream can be compressed while recording to conserve storage.

Bibliography

[1] M. C. Baur. Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs. Master's thesis, Formal Methods Group, Swiss Federal Institute of Technology (ETH, Zurich), April 2003.

[2] R. Girimaji. Reactor: A Software Pattern for Building, Simulating and Debugging Distributed and Concurrent Systems. Master's thesis, ITTC, University of Kansas, 2003.

[3] S. Halbhavi. Thread Debugger Implementation and Integration with the SmartGDB Debugging Paradigm. Master's thesis, ITTC, University of Kansas, 1995.

[4] H. Srinivasan J. Choi. Deterministic Replay of Java Multithreaded Applications. Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, August 1998.

[5] J.M. Stone J. Choi. Balancing runtime and replay costs in a trace-and-replay system. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 13 22, May 1991.

[6] D. Niehaus J. James. Correctness of a Programming Environment Supporting Reproducible Concurrency. Submitted to IEEE Transactions on Parallel and Distributed Systems.

[7] S. Mallavadi. A Thread Debugger for Testing and Reproducing Concurrency Scenarios. Master's thesis, ITTC, University of Kansas, January 2003.

[8] S. Penumarthy. Design and Implementaion of a User-Level Thread Library for Testing and Reproducing Concurrency Scenarios. Master's thesis, ITTC, University of Kansas, December 2002.

[9] K.C. Tai R. Carver. Deterministic execution testing and debugging of concurrent programs. Proceedings of the Pacific Northwest Software Quality Conference, pp. 170-182, 1989.

[10] J.C. King S.L. Hantler. An Introduction to Proving the Correctness of Programs. ACM Computing Surveys (CSUR), Volume 8 , Issue 3, pp. 331 - 353, September 1976.

[11] M.V. Zelkowitz. Perspectives in Software Engineering. ACM Computing Surveys (CSUR), Volume 10 , Issue 2, pp. 197 - 216, June 1978.

Appendix A

Important functions and files

Some of the important functions in Clever Insight are listed below. All filenames are referenced relative to the top-level directory of Clever Insight source.

Function name	*Task*	*Source filename*
setNextReplayBreakpoint	Sets up the next replay breakpoint from the event history	gdb/breakpoint.c
commands_tcl_command	Executes the tcl commands attached to a breakpoint	gdb/breakpoint.c
signal2give_command	Gives the signal specified in the script to the program	gdb/breakpoint.c
disable_last_breakpoint_command	Disables the last hit breakpoint	gdb/breakpoint.c
activate_replay_command	Activates the replay mode	gdb/infcmd.c
automate_replay_command	Automates the replay mode. The program continues till it completes or till it is interrupted.	gdb/infcmd.c
controlled_replay_command	Sets the replay mode to controlled. The user gets the control after a replay breakpoint is hit.	gdb/infcmd.c

Table A.1: Important functions and files

Appendix B

DSUI Configuration files

This section lists the various files used for configuring DSUI. These include the advanced_enable.dsui file (for enabling event stream recording) and the namespace files (for specifying events and families).

Listing B.1 The advanced_enabled.dsui file

```
DSTRM_EVENT             GCC             1       EVENT_BB_ENTER
DSTRM_EVENT             MAIN_ARGS       2       EVENT_APP_ARGS
DSTRM_EVENT             BTHREAD_LIB     3       EVENT_SCHED
DSTRM_EVENT             IO_WRAP         4       EVENT_IO_CLOSE
DSTRM_EVENT             IO_WRAP         4       EVENT_IO_OPEN
DSTRM_EVENT             IO_WRAP         4       EVENT_IO_READ
DSTRM_EVENT             IO_WRAP         4       EVENT_IO_WRITE
DSTRM_EVENT             LIB_INIT        5       EVENT_DUMMY
DSTRM_EVENT             SIGNAL          6       EVENT_SIGNAL
DSTRM_EVENT             REACTOR_WRAP    7       EVENT_FD_SET
DSTRM_EVENT             SOCKET_WRAP     8       EVENT_SKT_SOCKET
DSTRM_EVENT             SOCKET_WRAP     8       EVENT_SKT_ACCEPT
DSTRM_EVENT             SOCKET_WRAP     8       EVENT_SKT_RECV
DSTRM_EVENT             SOCKET_WRAP     8       EVENT_SKT_SEND
```

Listing B.2 The DSUI namespace file

```
##########################################
# Datastream Namespace Specification File #
##########################################
# EVENT ZEROES ARE DUMMY
##########################################
# ENTITY_TYPE  FAMILY_NAME  FAM_ID  ENTITY_NAME  ETY_ID  DESCRIPTION


#######################
# GCC EVENTS  #
#######################
DSTRM_EVENT  GCC  1  EVENT_BB_ENTER  0  "Entered a basic block event"

#########################
# APPLICATION ARGUMENTS EVENT1   #
#########################
DSTRM_EVENT  MAIN_ARGS  2  EVENT_APP_ARGS  0  "Record main Arguments"

#######################
# BTHREAD EVENTS  #
#######################
DSTRM_EVENT  BTHREAD_LIB  3  EVENT_SCHED  0  "A context switch event"

#######################
# BTHREAD WRAPPERS I/O EVENTS  #
#######################
DSTRM_EVENT  IO_WRAP  4  EVENT_IO_CLOSE  3  "close system call"
DSTRM_EVENT  IO_WRAP  4  EVENT_IO_OPEN  2  "open system call"
DSTRM_EVENT  IO_WRAP  4  EVENT_IO_READ  1  "read system call"
DSTRM_EVENT  IO_WRAP  4  EVENT_IO_WRITE  0  "write system call"

#########################
# LIBRARY INITIALIZATION   #
#########################
DSTRM_EVENT  LIB_INIT  5  EVENT_DUMMY  0  "Library initialization"

#######################
# SIGNAL EVENTS  #
#######################
DSTRM_EVENT  SIGNAL  6  EVENT_SIGNAL  0  "A signal event"

#######################
# BTHREAD WRAPPERS I/O EVENTS  #
#######################
DSTRM_EVENT  REACTOR_WRAP  7  EVENT_FD_SET  0  "Stores the FD Set"

#######################
# SOCKETWRAPPERS I/O EVENTS  #
#######################
DSTRM_EVENT  SOCKET_WRAP  8  EVENT_SKT_SOCKET  3  "socket system call"
DSTRM_EVENT  SOCKET_WRAP  8  EVENT_SKT_ACCEPT  2  "accept system call"
DSTRM_EVENT  SOCKET_WRAP  8  EVENT_SKT_RECV  1  "recv system call"
DSTRM_EVENT  SOCKET_WRAP  8  EVENT_SKT_SEND  0  "send system call"
```