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

Rajiv Ramanasankaran

Masters Thesis Defense
University of Kansas, Lawrence
May 20th, 2004

Committee:
Dr. Jerry James (Chair)
Dr. Douglas Niehaus
Dr. Perry Alexander

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Introduction

- Debugging
  - Process of finding and removing bugs
  - Helps in understanding program flow
- Types of bugs

Bugs due to …	Diagnosed by
Syntax	Compiler errors
Runtime	Exceptions
Logic/Design	Highly Variable
Temporal environment	Highly Variable

Introduction (contd…)

Temporal Environment

The sequence of states of a program consisting of the contents of all of its variables and its status registers (such as the program counter, the stack pointer).

The states change due to program source or due to scheduling decisions.

Introduction (contd…)

Factors affecting the Temporal Environment

- Multiple threads of execution
- Non-determinacy: repeated executions of the same program may give different results!
    - Scheduling of processes/threads
    - Signal delivery
    - I/O operations

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Motivation

- Reproduction of concurrency scenarios is a hard problem
- Limitations of traditional debugging techniques
  - Print statements, trace debugging and user-controlled breakpoints
- Absence of a framework for replaying execution by re-creating the temporal environment
- Insufficient access to the concurrency model while debugging
- Need for a definitive framework for testing and creating concurrency scenarios

Motivation (contd…)

- Work done at ITTC
  - BERT
    - Reactor pattern: Event demultiplexing framework
    - BThreads: User-level thread library
  - Clever Insight
    - Context switches
    - Signals
    - Interfaces with BThreads using the TDI (Thread debugger interface)
  - Need for a framework that replays software that performs I/O

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Solution

- Extend BERT, Clever Insight
- Reproduce the temporal environment
- Record and Replay I/O
- Develop a non-intrusive framework
- Allow creation of new concurrency scenarios
- Automate the framework

Outline

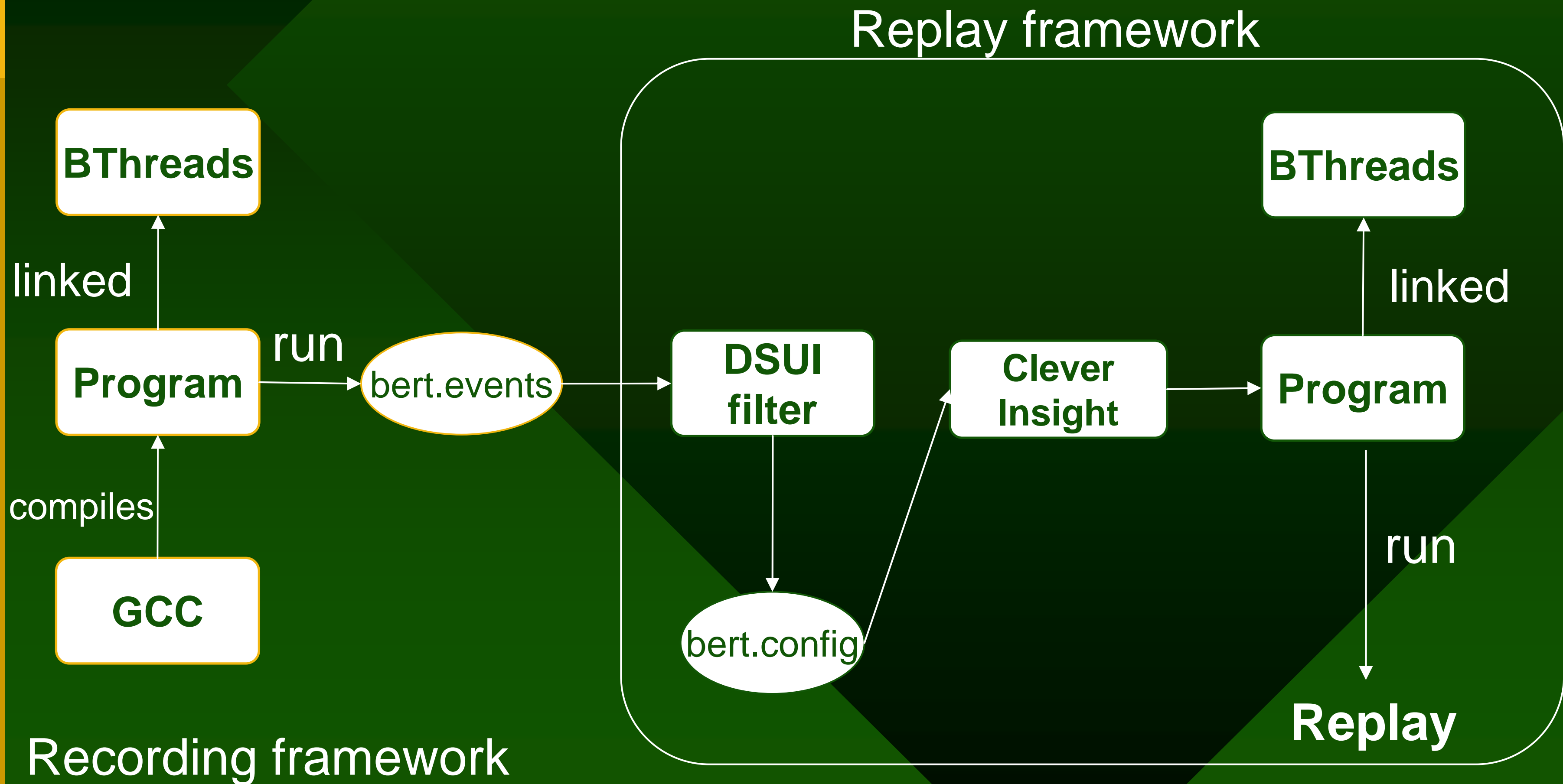
- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Parts of the framework

- Recording Framework
  - GCC: Recording a trace of basic blocks
  - BThreads: Recording context switches, signals and I/O events *(recording mode)*
- Replaying Framework
  - Clever Insight (context switch, signal delivery events)
  - BThreads: I/O events *(replay mode)*
- Data Stream User Interface (DSUI)
  - Tool for collecting data and trace information from a program
  - Ability to categorize records into events and families
  - Provides post-processing tools for parsing the recorded trace

Design and Implementation

System Model



Recording Framework

Recording the trace of basic blocks

- GCC provides a basic block profiling feature (*-ax command line option*)
- A *-bert* option is devised which generates a trace of basic blocks
- Helps in testing replay

Recording Framework

Recording I/O (BThreads)

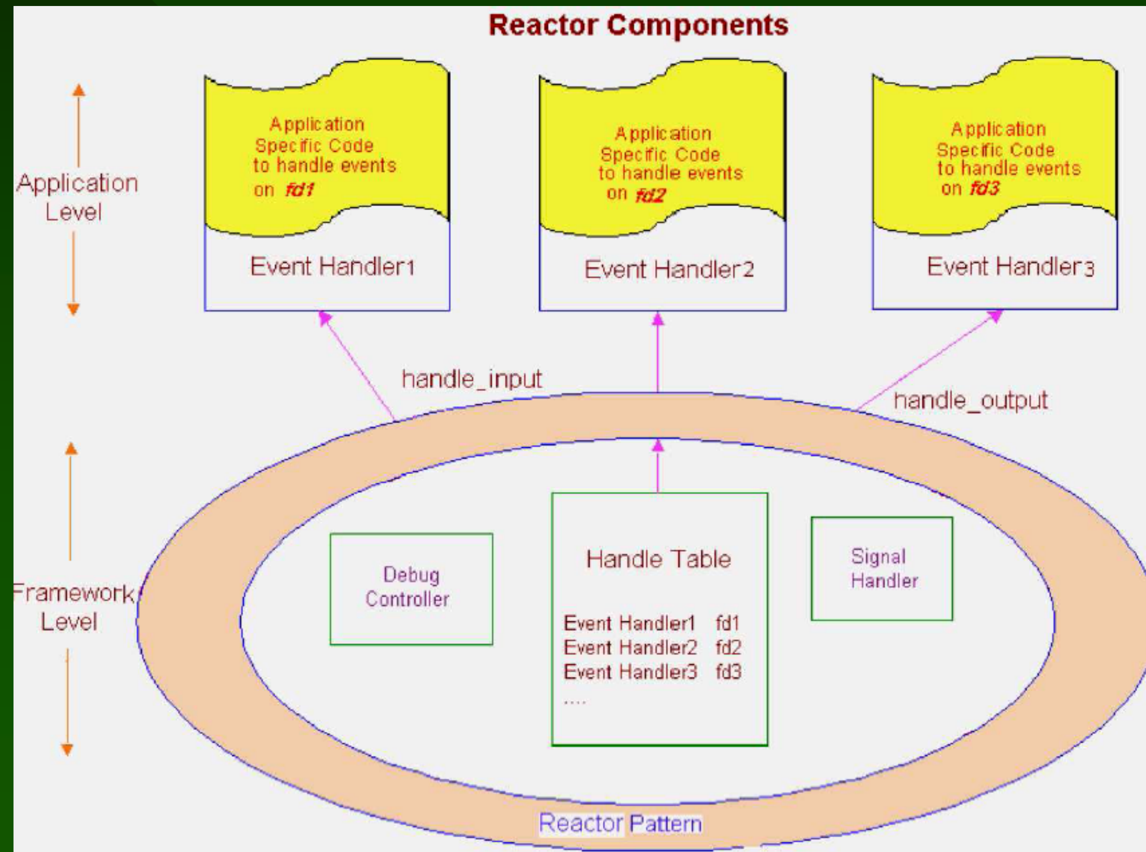
Event types:

- Reactor Events
  - Involves recording the list of registered handlers and their states in the Reactor
- I/O Events
  - Involves recording the contents of I/O buffers in a system call, return values and the effects (e.g., *errno*)
- Command-line arguments
  - Involves the recording of the command-line arguments given to the program

Recording Framework

Reactor Events

Understanding the Reactor and its event handlers



Event handlers

- handle_input
- handle_output
- handle_close

Recording Framework

I/O events inside BThreads

- Record the contents of the variables in a system call
  - e.g., `read ( int fd, void *buf, size_t count)`
- Record the return values of the system call
  - e.g., return values of read and write
- Record the effects of the system call
  - e.g., the *errno* variable

Command-line arguments

- Recording the arguments given to *main*

Recording Framework

Recording context switch, signal delivery events

Involves recording of

- The thread id
- Program counter (PC)
- The basic block number
- Count of the basic block
- Signal received

When the program is run, all the above events are directed to a single event history file.

Replay Framework

Post-processing using DSUI filters

- ***bert.events*** file contains the event history
- A configuration script is generated using a special DSUI filter
- Serves as a single input for Clever Insight to replay the program
- Contains the commands needed to set up the Replay framework

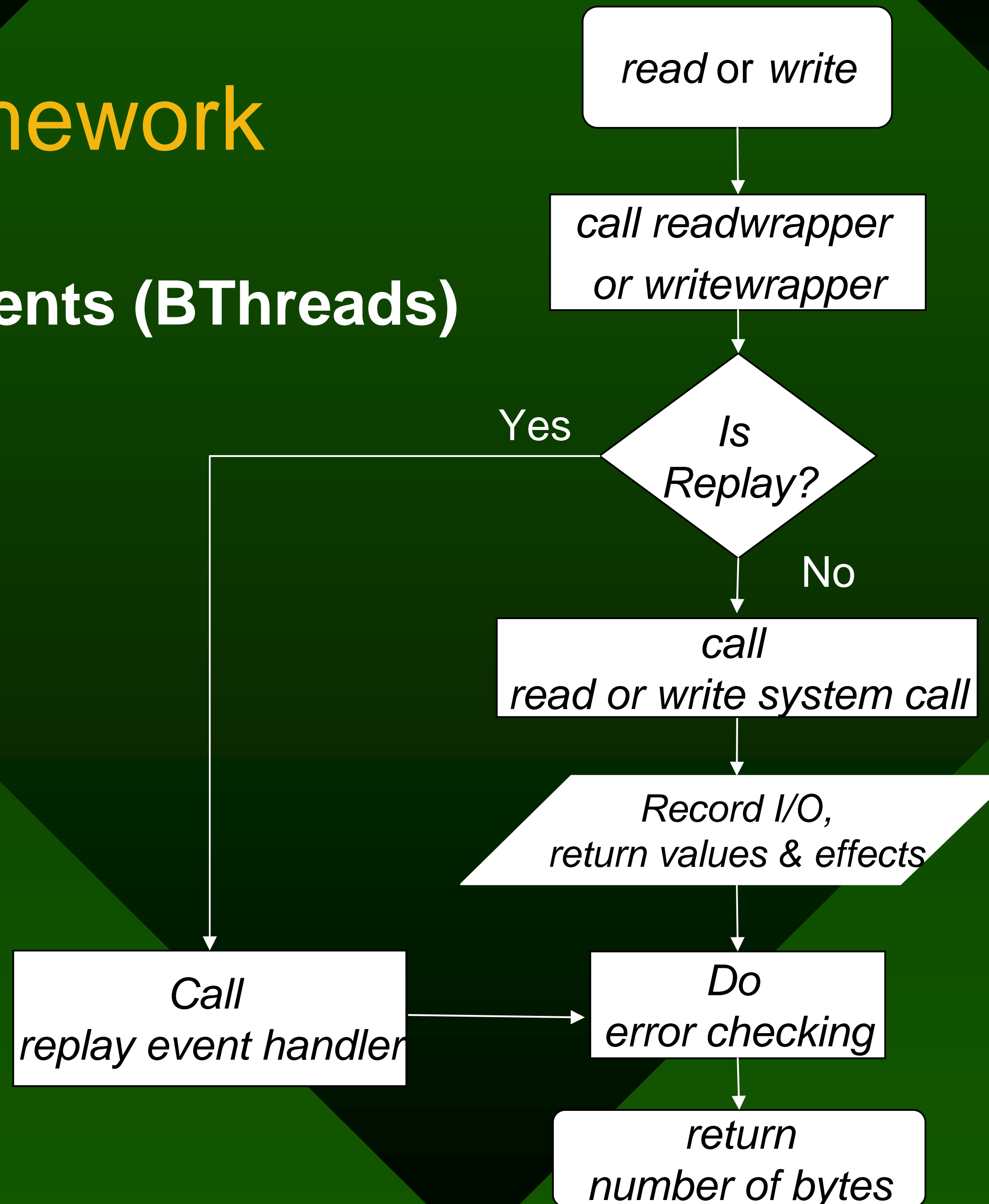
Replay Framework

Replay event handlers for I/O

- Instead of making the system calls, BThreads, in *replay* mode, calls the replay event handlers
- A replay event handler
  - fetches the recorded information from the event history; and also
  - copies the information to the system call buffers and variables which hold application state information (e.g., *errno*, return values)

Replay Framework

Replaying I/O events (BThreads)



Replay Framework

Replaying Reactor events (BThreads)

- **Recorded history contains**
  - a list of handlers;
  - handler types and
  - handler states (i.e. whether input or output could be performed without blocking)
- **When replaying**
  1. Iterate over the state of the handlers from history
  2. Call the appropriate function (handle_input or handle_output)
  3. State of threads are automatically restored since the handlers were processed in the same order

Replay Framework

Replaying context switches, signals (Clever Insight)

- Signal SIGPROF signifies the expiry of the scheduling timer and forces a context switch.
  - Before replay starts

```
handle SIGPROF nopass
```

- Set conditional breakpoints incrementally at places where there was a context switch or signal delivery

```
break <pc> if block==<blk> && count==<cnt>
```

contd…

Replay Framework

Replaying context switches, signals (contd…)

- When the breakpoint is hit, the signal (SIGPROF or other) needs to be delivered
  - **Controlled replay:**
    - The user gets control after every breakpoint; and
    - Signal must be delivered manually in order to continue replay
  - **Automated replay:**
    - The signal is automatically delivered each time;and
    - replay continues until the user interrupts it
  - These features have been implemented for both : Clever insight command-line mode as well as GUI mode

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Features provided

- *(de)activate_replay :* deactivates/activates use of Clever Insight as a replay tool
- *controlled_replay* command: replays program in controlled mode
- *automate_replay* command: replays program in automated mode
- *runtcl_later <tcl_script>*: executes the commands in the Tcl script
- *continue_replay* command: continues replay of the program in automated mode
- Attach arbitrary Tcl scripts to breakpoints

Features provided

- The user is free to use debugging primitives without disturbing replay
- Constructing new scenarios: Ability to replay the program till a point of interest (using controlled or automated replay) and
  - let the program continue (`handle SIGPROF pass`)
  - create a new concurrency scenario
- Switch to the desired thread (feature in Clever Insight)

```
call switchtothread (thread_id)
```

- Use the recorded history of the new scenario for further study

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Evaluation

- **Event history comparison**
  - **Programs tested: pc.c, diningphil.c, mmult.c, copy.c**
  - Compared the recorded basic block trace with the trace of the program being replayed
  - The history during replay conformed to the recorded history
  - Proves program took the same path while replay

Evaluation (contd…)

- **Testing I/O reproducibility**
  - copy.c :A multithreaded program to make multiple copies of a single large file was written and the event history is obtained
  - During replay, the calls to read the original file are directed to the event stream but writes to the copies of the file are allowed to proceed
  - The copied files are compared to the original and found to be equal in content

Evaluation (contd…)

- **Testing thread interleaving**
  - The dining philosopher’s program is executed
  - Specific thread interleaving leading to a deadlock is noted
  - The replayed program’s thread interleaving is compared to the original and found to be same

Evaluation (contd…)

- **Testing creation of concurrency scenarios**
  - The dining philosopher’s program is executed till a point of interest but before any deadlock takes place
  - Thread states are changed using Clever Insight to force a deadlock
  - The new recorded event history is obtained and replayed
  - The replayed program reaches a deadlock as per the scenario that was created

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Related work

- Deterministic execution testing of ADA programs
  - Uses source level transformations to generate a file of synchronization events when the program is run
  - This file is then used to test execution with different inputs
  - Needs specific transformations for different synchronization events
  - Intrusive framework needing a lot of edits and re-compiling for testing different scenarios
  - No support for I/O reproducibility

Related work

- DejaVu
  - Modified JVM
  - Captures thread schedule and accesses to shared variables
  - Reproduces execution but no support for I/O
  - Changes made to the JVM are specific to the synchronization mechanisms
  - No support for creating new scenarios

Related work

- JReplay: Instrumenting Java Bytecode
  - Innovative tool doesn't modify the JVM. Instead patches the compiled Java class files
  - Transforms a nondeterministic multithreaded program to a sequential deterministic one by locking all threads but one
  - Surrounds thread operations with locks in bytecode controlled by an external scheduler and a recorded thread schedule
  - Depends heavily on the format of the Java class bytecode
  - Deprecated Java API poses problems
  - Doesn't support I/O reproducibility

Outline

- Introduction
- Motivation
- Solution
- Design and Implementation of the Framework
- Features provided by the Framework
- Evaluation
- Related work
- Conclusions and Future work

Conclusions

- Formalized a process of recording a program’s history and replaying it by re-creating the temporal environment
- Created a non-intrusive framework
- The framework exposes the concurrency model to the user, yielding full control over it
- Supports traditional debugging primitives and offers a choice to the user
- Supports the ability to create new scenarios
- Introduced a new method of interactivity through the controlled and automated replay modes

Future work

- Extend the framework to distributed systems by simulating the network on top of BERT
- Reproducible executions of Java programs by porting the JVM on to BERT
- A richer set of GUI widgets to make debugging and creation of concurrency scenarios easier
- Cover replay of more program types by writing wrappers for other system calls
- Reproduce executions of the operating system by porting User-mode Linux on to BERT
- Compress the event history to conserve space

Thank you