COMPUTER SCIENCE DEPARTMENT
INDIAN INSTITUTE OF TECHNOLOGY

BTECH PROJECT

Record/Replay Debugger (RRDebug)

Author:
Ankur DAHIYA

Supervisor:
Prof. Sorav BANSAL

May 2012

Abstract

RRDebug is a debugger based on the Record and Replay functionality of Qemu/KVM. It takes as input the RR-log of a virtual machine. The exact execution of the VM can be reconstructed using this log.
RRDebug replays the machine and logs all memory effects, i.e. all the instrucions that modify memory. All these entries are stored in a SQLite database which can then be queried. We have also developed a prototype Query UI.
This report outlines the design and development of RRDebug.

Contents

Chapter 1

Introduction

We have developed a tool called RRDebug in the two semesters as part of my B.Tech. Project. The source code of the project can be accessed at `https://github.com/legalosLOTR/btp`.
This report describes the goals, developement and the final result of this project.

1.1 Organisation of the report

The following sections present an introduction to the project. First we describe what is Record and Replay, and then the motivation and goals for RRDebug.
The second chapter discusses the design of RRDebug and the challenges faced during development.
The third chapter gives a brief overview of how the system works.
The fourth chapter presents some performance metrics.
The fifth chapter presents a conclusion.
The sixth chapter outlines some focus areas for future developers of this project.
The seventh chapter has a list of all the references used within this text.

1.2 Record and Replay

Record and Replay is a feature developed by Prof. Bansal and others for Qemu and KVM [1]. During record, it logs all non-deterministic effects like IO, interrupts, network packets into a log-file. This file can then be replayed to exactly reconstruct the execution. Thus, all bugs that were encountered during the recording can be replayed and studied as many times as required.

1.3 Motivation

Debugging large pieces of software (like OSes, Web browsers etc.) is a very difficult task. A major cause of concern is fixing concurrency bugs. These bugs can arise due to improper synchronization. Even if such a bug is discovered, replicating it can be very time-consuming!
This is the exact problem that RRDebug is trying to solve. We have a system (Record and Replay) that allows us to record the execution of a VM, with minimal overhead[1]. This recording can then be replayed and debugged. This debugging can be done in real time (i.e. when replaying). But this is not very efficent as for each query, the whole replay has to be executed. Another approach is to preprocess the recording in some way and generate a database which allows us to answer queries more efficiently, without replaying. RRDebug allows both modes. It can either be run in real time mode or it can run the replay once and build a database for answering queries.
There is another system, called Chronicle-Recoder [2], which has similar goals. It also builds a database to answer such queries. But it has a very large overhead during runtime and thus, cannot capture all the real-life concurrency bugs.

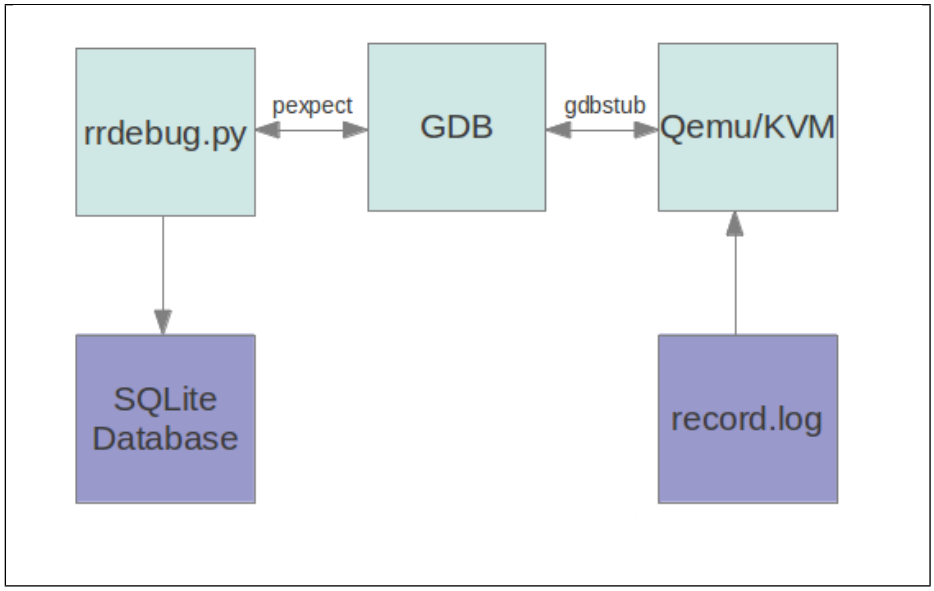
1.4 Goals

- Develop a debugging framework based on Record and Replay
- Determine an efficient way to answer the query “Where was this variable modified?” (Focus on kernel variables only.)

- Design an efficient format for the database generated by the record pre-processing.
- Develop an intuitive GUI that allows a user to query the database and get answers.

Chapter 2

Design



The main part of the debugger is a python program (rrdebug.py). This program interacts with Qemu/KVM and GDB to accomplish its goals. To run properly, it needs some modifications to the Record and Replay (RR) code. We have developed a patch that can be applied to the RR code (at revision-80) to achieve this.

2.1 Libraries and tools used

- **GDB** to read memory/registers from Qemu/KVM

- **pexpect** library to interact with GDB from python
- **sqlite3** for the database

2.2 Real Time Debugger

The real-time component of RRDebug allows the user to query for any user/kernel variable. RRDebug pauses the execution at every place where that particular variable is modified and prints the backtrace and the variable's value.
RRDebug uses GDB to set HW watchpoints on the respective memory locations. When these watchpoints are hit, the Debugger pauses execution and prints the backtrace. This works for both Qemu and KVM.

2.3 Indexing engine

For the indexing engine to work, we need to trace each instruction and log those that affect memory locations. We made some changes to the RR code which now allows gdb to toggle kvm's single-stepping mode. Thus, we can instruct kvm to return control to gdb after each instruction. The instruction is then disassembeld by gdb and interpreted by RRDebug (explained in the next section) to determine the affected memory region. Once this instruction is executed, we check the memory region for any changes and log them. Thus, we are able to log all memory effects to our database.
The only downside of this approach is that it leads to a tremendous slowdown as kvm has to return to user mode after each instruction. But this is not a major issue as the indexing procedure has to be done only once. Once the database is ready, we dont need to replay or index ever again.

2.3.1 Interpreting Instructions

Given the disassembly of an x86 instruction, we need to determine the memory region that this instruction might affect.
The destination memory address is determined from the registers or the immediate

values present.
Right now, the system handles only the frequent (and simpler) instruction like mov etc.

2.4 Database

As stated earlier, RRDebug employs a SQLite database. SQLite is suitable for our purpose as it does not need any server (like MySql) and the whole database engine and the data itself fits in a single file!
The database schema consists of the following fields:

- EIP
- Memory Address
- Old Data
- New Data
- Backtrace, at point of occurence of event
- Timestamp, when this event was logged

The database is indexed by the *Memory Address* field, as our queries will always be targetting a variable (which can be converted to a memory address using the vmlinux file).
We can eliminate the *Old Data* field by taking a dump of the whole memory at the start of the indexing. But it has been kept in this version for simplicity sake.

2.5 Query Engine

RRDebug also includes a GUI Query Engine built using python/wxPython. It reads the SQLite Database built by the indexer and allows the user to run queries against it. The queries can include variable names, which are translated to physical addresses using GDB (via pexpect). The search function works in both directions - forward and reverse.

2.6 Screenshots

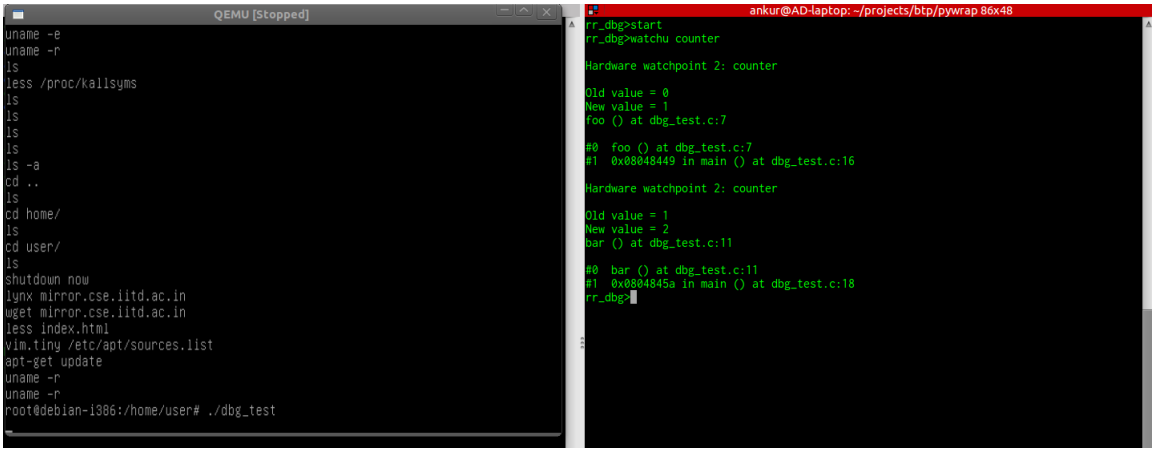


Figure 2.1: Screenshot of RRDebug along with qemu

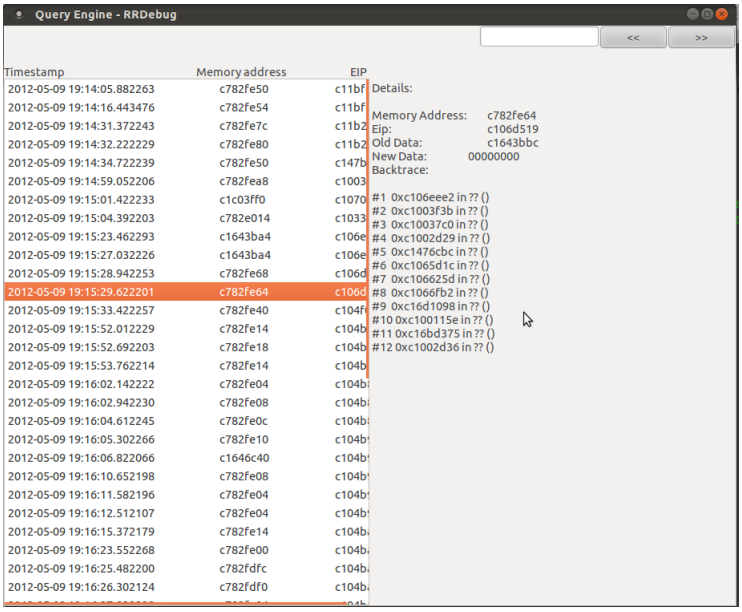


Figure 2.2: Screenshot of RRDebug Query Engine

Chapter 3

Conclusion

We set out to develop a debugger that can efficiently take advantage of the Record and Replay feature. RRDebug achieves this by allowing a user to do real time debugging while replaying as well as a much more efficient index based debugging. Also, we have determined that gdb can be used to access and debug a VM replay in a very convineint way. This knowledge can be of help to others who work with the RR system in the future.

Chapter 4

Future Work

- Improve the Query Engine UI.
- Add more instructions to the Indexing engine.
- The indexing process can be sped up if we turn off single step mode when there are no relevant instructions around. This can be achieved using some appropriate heuristics.
- Perform more rigourous tests.

Bibliography

[1] Sorav Bansal, Piyus Kedia; Milisecond Rollbacks to recover from Software Failures

[2] Robert Callahan; Chronicle Reader, Valgrind based indexed execution recording; *http://code.google.com/p/chronicle-recorder/*