Replay Debugging: Leveraging Record and Replay for Program Debugging

Nima Honarmand and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu/
Motivation: Bug Reproduction is Difficult

Especially for bugs in production runs

Due to

- Complex inputs
- Non-deterministic timing in concurrent programs

**Record and Deterministic Replay (RnR)** can help

- **Recreates** execution of a program
- **Record**: capture non-deterministic events in a log
- **Replay**: use the log to recreate the exact same execution

**Problem**: Current RnR solutions are not quite suitable as debugging tools.
RnR Logs in QuickRec [ISCA’13]

Input Log: Program/OS interactions
- System call results
- Data copied to application buffers by OS kernel
- Signals, ...

Memory Access Interleaving Log
- Inter-thread dependences

Chunk-based Recording

Captured in OS kernel

Captured using special HW

“Replay Debugging”
Nima Honarmand and Josep Torrellas
Problem: Naïve RnR not Enough for Debugging

Replay only reproduces the buggy execution

Log
Program

X Crash

Not enough! need to augment the code for diagnosis

Augmented code cannot be replayed using the recorded log:
– Different input events
– Different sequence of insts

Replay Debugging:
Augment the code and still be able to replay the log

Log
Program

X Crash

Bug Found !!!

Debug code 3
Debug code 2
Debug code 1

Enables fast & deterministic convergence to the source of the bug
Effective Bug Diagnosis

...Needs the ability to

- Write debug code as if part of the same program (may be inlined with main code)

```c
int a = /* program code */;
#ifdef DEBUG
    printf("a is %d", a);
#endif
```

- Access main program state
- Call main program functions
- Output results of debug code
- Have debug-only state in the debug code, e.g.,
  - Local and global variables, heap-allocated objects, shadow data structures

How to enable all of this without breaking replay?
Contribution: Replay DeBugging (RDB)

• A methodology for guaranteed deterministic replay in presence of debug code
  → One can debug a non-deterministic bug deterministically

• A design combining compiler technology + replay mechanisms
  • Implementation using LLVM and Intel’s Pin

• Seamless debugging experience for the programmer
  − RDB debug code very similar to ordinary debug code
**RDB Approach: Overall View**

1. **Writing Debug Code**
   - Programmer adds debug code to program source code

2. **Extracting Debug Code**
   - Compiler extracts debug code and generates **two binaries**
     - One contains original, unmodified code
     - Other contains the debug code

3. **Executing Debug Code While Replaying**
   - Replay tool automatically invokes debug code at correct points while replaying the log

**Modified LLVM Compiler**
1. Programmer Writes Debug Code

Debug Code:

Should be marked using special markers

- Can read main program state
- Can invoke main program functions
- Should not write to main program state (directly or indirectly)

Can have its own state (local, global and heap)

Use the same virtual address space as the main code
- E.g., debug vars can point to main data

Can use runtime library functions
- E.g., printf() or malloc() from libc
- Will have its own instance of runtime libs during replay
2. Compiling the Augmented Code

**Front-end** creates unoptimized LLVM IR from source code

**Optimizer** transforms LLVM IR to optimized form
- We assume all optimizations are disabled for now

**CodeGen** generates machine code

We modify the last two
2.1. Generating Initial LLVM IR

C/C++ Code → Clang Front-end → LLVM IR-level Transformations → LLVM IR → LLVM CodeGen Backend (x86) → Machine Code

No Changes to the Front-end

void main(void) {
    char c;
    c = getchar();
rdb_begin
printf("c is '%c'\n", c);
rdb_end
}

@.str = "c is '%c'\n"

void @main() {
    %c = alloca i8
    %tmp0 = call @getchar()
    store %tmp0, %c
    call @__rdb_begin()
    %tmp1 = load %c
    call @printf(@.str, %tmp1)
    call @__rdb_end()
}
2.2. Extracting the Debug Code

Extract the debug code

- **C/C++ Code**
  - Clang Front-end
  - LLVM IR
  - LLVM IR-Level Transformations
  - LLVM CodeGen Backend (x86)

- **LLVM IR (from Front-end)**

- **LLVM IR-Level Transformations**

- **Extracted Main Code (LLVM IR)**

- **Extracted Debug Code (LLVM IR)**

- **Function Descriptors (C++)**

```
@.str = "c is '%c'\n"

void @main()
{
  %c = alloca i8
  %tmp0 = call @getchar()
  store %tmp0, %c
  call __rdb_begin()
  %tmp1 = load %c
  call @printf(@.str, %tmp1)
  call __rdb_end()
}
```

```
void __rdb_func_1(i8* %arg)
{
  %tmp1 = load %arg
  call @printf(@.str, %tmp1)
}
```

```
void @main()
{
  %c = alloca i8
  %tmp0 = call @getchar()
  store %tmp0, %c
  call @llvm.rdb.location(1)
  call @llvm.rdb.arg(1, 0, %c)
}
```
2.3. Generating Machine Code

C/C++ Code \rightarrow Clang Front-end \rightarrow LLVM IR \rightarrow LLVM IR-level Transformations \rightarrow LLVM CodeGen Backend (x86) \rightarrow Machine Code

Extracted Main Code (LLVM IR)

void @main() {
  %c = alloca i8
  %tmp0 = call @getchar()
  store %tmp0, %c
  call @llvm.rdb.location(1)
  call @llvm.rdb.arg(1, 0, %c)
}

Extracted Debug Code (LLVM IR)

void @_rdb_func_1(i8* %arg) {
  %tmp1 = load %arg
  call @printf(@.str, %tmp1)
}

@.str = "c is '%c'\n"

`\texttt{FuncID}
\begin{tabular}{cccc}
\hline
\text{Position} & \text{Class} & \text{Info} \\
\hline
1 & 0 & Stack & (SP, -20) \\
2 & 0 & ... & ... \\
2 & 1 & ... & ... \\
\end{tabular}`

“Replay Debugging”
Nima Honarmand and Josep Torrellas
3. Replay Tool Invokes Debug Code

Replay implemented using Intel’s Pin (similar to QuickRec)

- A binary instrumentation infrastructure

Anatomy of Pin

- Program and Pintool in the same address space
- Pintool is use-case specific

Our pintool, RdbTool, does two things:

- Replays the log
- Invokes debugging code
3. Replay Tool Invokes Debug Code

To replay, RdbTool
- Instruments system calls to inject program inputs
- Counts # of insts to enforce recorded interleaving

To invoke debug code, compile debug code into RdbTool

RdbTool then
- Sets breakpoints at debug markers
- Finds and invokes debug code using Function and Argument descriptors
3. Replay Tool Invokes Debug Code

Loads the main code; links it with runtime libraries

Loads the RdbTool; links it with separate runtime libraries

Replays the main code & invokes debug code on hitting a debug marker

Execution is the same as recorded in the log
Problem with Compiler Optimizations

Optimizations will be performed after extracting debug code

May render the debug code invalid
  – E.g., may optimize away state needed by the debug code

Work in progress…

```
void f() {
  char c = getchar();
  int a = c ? 5 : 6;
  printf("c is %d\n", c);
  rdb_begin
  printf("a is %d\n", a);
  rdb_end
}
```
Also in the Paper

- Real example of bug diagnosis with RDB
- Support for event-driven debugging (watch points)
- Enforcing read-only access to main-program’s memory
- Using gdb together with RDB
- Replay debugging without Pin
- ...

“Replay Debugging”
Nima Honarmand and Josep Torrellas
Conclusions

Naïve RnR not enough for bug diagnosis

Replay Debugging: A methodology for guaranteed deterministic replay in presence of debug code

• Seamless debugging experience for programmer
• Combines compiler and replay technology
• Proof-of-the-concept implementation using LLVM and Pin

With RDB, one can diagnose a non-deterministic bug deterministically
THANK YOU!