Replay Debugging

Leveraging Record and Replay for Program Debugging (ISCA 2014)
- Nima Honarmad and Josep Torrellas
  (University of Illinois at Urbana-Champaign)
Introduction

● This Work...
  ○ Uses RnR (Record and Replay) for enabling debugging by adding your own debug code while guaranteeing exact replay!

● This work does not...
  ○ Build a RnR system

*Traditional RnR only guarantees exact replay*
Record and Replay

● Input Recording
  ○ Sources of non-determinism
    ■ Program’s interfaces with OS (program inputs)
    ■ Memory race between threads

● Replaying
  ○ Injecting recorded inputs at correct times
  ○ Enforcing inter-thread dependencies
Input log

- Captures all program’s interfaces with OS
  - System call return values
  - Data copied to application buffers by the OS
  - Signals
  - Results of some non-deterministic processor instructions (rtdsc)

- Log sizes are typically of order 1.2KB per million instructions (?)
Memory Race Recording

- Special hardware on each core
- Idea is to divide thread’s execution in chunks
- Record the order between chunks
Memory Race Recording

Thread 1:
- %r1 = a
- %r2 = b
- t = %r1 + %r2

Thread 2:
- %r1 = t
- %r2 = c
- %r3 = %r1 + %r2

Core 1:
- (1) T1:2
- (1) T2:1
- (2) T1:1

Core 2:
- (2) T2:2

Chunk Log:
- T1:2
- T2:1
- T1:1
- T2:2
Organization of QuickRec

Application

OS

CPU + MRR

input log

chunk log

Application

Pin
Debugging breaks replay

- For debugging programmers perform things like
  - Inspect program state: registers/variables/memory
  - Calculate expressions based on state
  - Add prints
  - Create and keep debug state globally / on stack / on heap

*Unfortunately, this distorts program replay...*
Debugging breaks replay

- Why replay gets distorted?
  - Incorrect chunk boundaries
  - Different set of system calls
Supported debugging features

- Inline debug code in the program code
  - Challenge: Adding debug code may render RnR log obsolete
  - Solution: Add a compiler pass to extract debug code from the main program
    - Debug code is put in another binary that shares address space with the main program
Supported debugging features

- Access program code, data from debug code
  - Challenge: *Sharing address space of debug program with main program*
  - Solution: Place debug code in areas that are not used by main program
    - RnR input log comes to the rescue!
Supported debugging features

- Output the results of the debug code
  - **Challenge:** Calling `printfs` would change runtime library state
  - **Solution:** Provide the debug code with its own instance of runtime library (e.g. libc / libstdc++)
Rdb base design

● Structure of debug code

```c
if(...) {
    N = ...; /* program code */
    x = ...; /* program code */
    rdb_begin
    int i;
    for(i = 0; i < N; i ++)
        printf("x[%d] = %d
", i, x[i]);
    rdb_end
}
```

● Rdb generates 2 executables: 1. identical to the original program, 2. containing extracted debug code
Rdb compilation flow

Clang Front-End → LLVM IR → LLVM IR-Level Transformations → LLVM IR → LLVM CodeGen Backend(x86)

Clang Front-End

Extractor Pass

Main Code + debug regions (LLVM IR)

Main Code

Extracted Debug Module (LLVM IR)

Main Module (LLVM IR)

Function Descriptors

CodeGen

Debug Code

CodeGen

Main Code + Location markers

Argument Descriptors
void main()
{
    char c;
    c = getchar();
    rdb_begin
    printf("c: %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()
}
Rdb compilation flow

Clang Front-End -> LLVM IR -> LLVM IR-Level Transformations -> LLVM IR -> LLVM CodeGen Backend(x86)

Extractor Pass
- Main Code
- Clang Front-End
- Main code + debug regions (LLVM IR)

Extracted Debug Module (LLVM IR)
- Main Module (LLVM IR)
- Function Descriptors

CodeGen
- Debug Code
- Main Code + Location markers
- Argument Descriptors
Rdb compilation flow

Clang Front-End → LLVM IR → LLVM IR-Level Transformations → LLVM IR → LLVM CodeGen Backend(x86)

Extractor Pass:
- Main Code
- Main code + debug regions (LLVM IR)

Extracted Debug Module (LLVM IR)
- Main Module (LLVM IR)
- Function Descriptors

CodeGen:
- Debug Binary
- Main Binary + Location markers
- Argument Descriptors
@.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()
}

@.str = "c is %c\n"
void @__rdb_func_1(i8 %arg) {
  %tmp1 = load %arg
  call @printf(@.str, %tmp1)
}

void @main() {
  %c = alloca i8
  %tmp1 = load %c
  call @__rdb_begin()
  %tmp0 = call @getchar()
  store %tmp0, %c
  call @llvm.rdb.location(1)
  call @llvm.rdb.arg(1, 0, %c)
  call @__rdb_end()
}

Function Desc
ID    FuncName
1    __rdb_func_1
2    ...

Rdb compilation flow

Clang Front-End → LLVM IR → LLVM IR-Level Transformations → LLVM IR → LLVM CodeGen Backend(x86)

Extractor Pass

Main Code

Clang Front-End

Extracted Debug Module (LLVM IR)

Function Descriptors

Main Module (LLVM IR)

CodeGen

Debug Code

Main Code + Location markers

Argument Descriptors
void @main() {
    %c = alloca i8
    %_tmp0 = call @getchar()
    store %_tmp0, %c
    call @llvm.rdb.location(1)
    call @llvm.rdb.arg(1, 0, %c)
}

pushq %rbp
...
call getchar
...
_leave_1:
leave
ret
Execution of debug code

- **Requirements**
  - Virtual address space sharing between debug code and main program
  - Ability to replay and enforce memory interleavings
  - Ability to invoke debug functions without affecting deterministic replay

- *rdb is built using ‘pin’*
Replay debugging using pin

- Address space of an application that runs under pin
Replay debugging using pin

- **Rdbtool**: a pintool built by compiling together:
  - The core logic of replay debugging
  - Object files of extracted debug code
  - Function and argument descriptor files
Execution

- Rdbtool sets breakpoints by looking up the symbol table
- Instruments system calls
- Chunk boundaries are ensured by keeping instruction count
- Uses function / argument descriptors to invoke debug code
Current limitations

● Adding / removing code in the main program
● Compiler optimizations
● Cross-region data sharing
Questions?