Replay Debugging: Leveraging Record and Replay for Program Debugging [*]

Nima Honarmand and Josep Torrellas
University of Illinois at Urbana-Champaign
{honarma1,torrella}@illinois.edu
http://iacoma.cs.uiuc.edu

Abstract

Hardware-assisted Record and Deterministic Replay (RnR) of programs has been proposed as a primitive for debugging hard-to-repeat software bugs. However, simply providing support for repeatedly stumbling on the same bug does not help diagnose it. For bug diagnosis, developers typically want to modify the code, e.g., by creating and operating on new variables, or printing state. Unfortunately, this renders the RnR log inconsistent and makes Replay Debugging *(i.e., debugging while using an RnR log for replay) dicey at best.*

This paper presents `rdb`*, the first scheme for replay debugging that guarantees exact replay.* `rdb` *relies on two mechanisms. The first one is compiler support to split the instrumented application into two executables: one that is identical to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime infrastructure that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. We describe an implementation of* `rdb` *based on LLVM and Pin, and show an example of how* `rdb`*'s replay debugging helps diagnose a real bug.*

1. Introduction

There has been substantial recent interest in the computer architecture community on hardware-assisted Record and Deterministic Replay (RnR) of programs (e.g., [5, 7, 17, 18, 19, 30, 31, 32, 33, 36, 37, 38, 39, 45, 46, 47]). This primitive consists of automatically recording the non-deterministic events of a program's execution in a log, and later using the log to replay the program deterministically. The typical non-deterministic events that are logged are the inputs to the program (such as system call return values and side effects, and signals) and the memory access interleavings of the threads.

One of the main usage models proposed for RnR is program debugging. The motivation is that some software bugs such as data races are often hard to repeat across executions with the same inputs, which makes them hard to debug. Hence, having the ability to deterministically reproduce an execution should help debug them.

However, simply providing support for repeatedly finding the same bug will not help remove it. The process of debugging involves modifying the program, for example by adding code to read program variables, create and operate on new variables, and print state out. We call the process of performing all of these operations while using a log to replay an execution *Replay Debugging*.

Unfortunately, any of these changes is very likely to distort the program's code and/or data, forcing the replayed execution to follow different paths than in the original execution encoded in the log. As a result, the log becomes inconsistent and cannot guide the new execution.

In practice, prior work has shown that this scenario still has some value. The relevant system, called DORA [43], can help diagnose bugs or test software patches. However, DORA often faces substantial divergence between the replayed execution and the original one. Importantly, it cannot guarantee deterministic replay. As a result, it is unable to ensure the exact reproduction of non-deterministic events that resulted in a bug. This is especially problematic when dealing with timing-dependent events like data races or atomicity violations.

Our goal, instead, is to be able to always *guarantee exact replay* during debugging, to quickly diagnose even highly non-deterministic bugs. We argue that, to guarantee replay debugging with exact replay, we need two capabilities. One is the ability to generate, out of the instrumented code, an executable that is identical to that of the original application. The second is the ability to replay the execution encoded in the log while invoking the debug code at the appropriate locations in the code.

To attain this goal, this paper presents `rdb`, the first scheme for replay debugging that guarantees exact replay. With `rdb`, the user interface is like in an ordinary bug diagnosis process. The user can read program variables, invoke program functions, create and use new debug variables and debug functions, set watchpoints, and print state. However, he cannot modify the state or instructions used by the program itself. Under these conditions, `rdb` uses the log generated by hardware-assisted RnR to guarantee deterministic re-execution.

`rdb`'s capability is possible thanks to two mechanisms. The first one is a compiler pass that splits the instrumented application into two binaries: one that is *identical* to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime infrastructure that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. No special hardware is needed beyond the original RnR system.

Overall, the contributions of this paper are:

- It presents `rdb`, the first scheme for replay debugging that guarantees exact replay.
- It describes an open-source implementation of `rdb` using

[*]This work was supported by NSF under grants CCF-1012759 and CNS-1116237, and by Intel under the Illinois-Intel Parallelism Center (I2PC).

LLVM [2] for the compiler mechanism and Pin [29] for the runtime mechanism.

- It discusses an example of how `rdb`'s replay debugging is used to diagnose a real bug.

This paper is organized as follows: Section 2 gives a background; Section 3 discusses how to use RnR for replay debugging; Sections 4–5 describe `rdb`; Section 6 presents an example of replay debugging; Section 7 outlines limitations; and Section 8 covers related work.

2. Background and Motivation

2.1. Hardware-Assisted Record and Deterministic Replay

To support RnR of a program's execution, all sources of non-determinism that can affect the execution need to be captured. Recent proposals for hardware-assisted application-level RnR consider two classes of non-deterministic events: *program inputs* (such as the results of system calls the program makes to the operating system (OS) or signals the program receives) and the *memory-access interleaving* of concurrent threads that result in inter-thread data dependences (in case of multithreaded programs). The process of recording the latter is sometimes called Memory Race Recording (MRR). Hardware-assisted RnR systems usually use OS support for input recording and special hardware for MRR. This is in contrast to software-only RnR solutions, which either do not support multithreaded programs or use the OS to record both inputs and memory races. In either case, the non-deterministic events are recorded in a log. Then, as we replay the execution, the system injects the recorded program inputs at the correct times and enforces the recorded interleavings to attain deterministic re-execution.

Almost all of the recent MRR techniques are based on the concept of *Chunks* of instructions (also called *Blocks* or *Episodes*). The idea is to divide each thread's execution into a sequence of dynamic groups of instructions or chunks. The execution of each chunk is logged as the number of instructions the chunk contained. The MRR hardware also records a partial or total order of all of the application's chunks. For each inter-thread data dependence, the chunk that contains the source of the dependence is ordered before the chunk that contains the destination. During replay, each chunk is executed after all of its predecessors (in the recorded order) and before any of its successors. In this manner, all inter-thread dependences are enforced. The replayer counts the number of instructions executed in a chunk to know when its execution is complete.

With no loss of generality, this paper assumes a hardware-assisted RnR environment like QuickRec [36]. QuickRec is an existing RnR prototype that uses OS support to record program inputs, and special hardware implemented with FPGAs for MRR. Recorded program inputs include system calls, data copied to application buffers by the OS as a result of system calls, signals, and results of some non-deterministic processor instructions. Memory interleaving is captured as a log of totally-ordered chunks.

QuickRec's replay tool is based on Intel's Pin [29] binary instrumentation infrastructure. It takes the application binary together with the recorded input and memory logs. As the program replays, it is able to inject the recorded inputs at appropriate points. In addition, it counts the instructions of each chunk as it executes, and enforces the recorded size and ordering of each chunk. Figure 1 illustrates the high-level structure of the system.

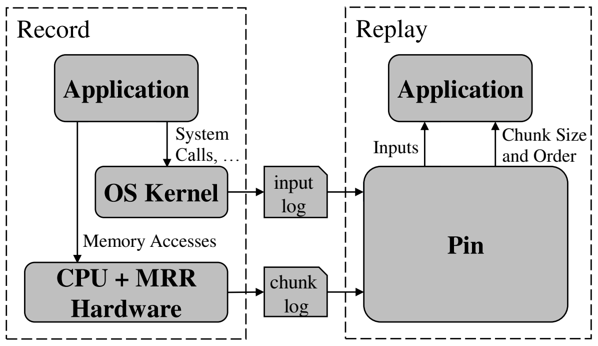


Figure 1: High-level organization of an RnR system.

2.2. Debugging Can Break Replay

To diagnose the root cause of a bug, programmers typically employ a process that involves the use of a debugger (e.g., gdb [1]), as well as writing some debug code. To do an effective job, programmers should be able to write code to perform at least the following tasks:

- Inspect program state, including registers, variables, and memory content of the program.
- Calculate expressions based on such state. This can involve calling subroutines from the program being debugged.
- Present the inspection results, e.g., using `print` statements.
- Create and keep debug state in the form of local, global or heap-allocated data structures used only for debugging.
- Set breakpoints and watchpoints to trigger some debugging activity when certain conditions become true.

Such a debugging process almost always involves distorting the code and/or data state of the program. Unfortunately, RnR mechanisms are very sensitive to such distortion. As a result, if we try to use the RnR log created by the original execution to replay the distorted program, we will observe replay divergence from the log.

Specifically, any changes in the code or data layout can potentially affect the control flow of a thread, changing the number and type of instructions executed. In hardware-assisted RnR, it causes the chunk boundaries to be placed at wrong instructions during replay, causing potentially-incorrect chunk orderings. This, in turn, can violate the recorded inter-thread data dependences.

In addition, code or data changes can cause replay divergence even in single-threaded programs, where memory access interleaving is not a concern. For example, code changes may result in a different set of system calls than was recorded, or system calls that are invoked with different operands. As another example, programs often use the value of pointers (i.e., addresses of variables) to construct data structures such as sets or maps of objects. If the pointer values change, the internal layout in such data structures will change. When the program traverses these data structures, changes in pointer values can result in different traversal orders and executions.

Viennot et al. [43] consider the problem of replaying modified programs in the context of a software-only RnR engine called SCRIBE [22] (as opposed to our focus on hardware-assisted RnR). SCRIBE uses OS support to record the non-deterministic events of an execution. Their proposed "mutable replay" system, called DORA, then uses a search-based technique to find and compare different ways of augmenting (or modifying) the recorded log in order to have it guide the execution of the modified code. When the search fails, DORA switches from replaying to recording in order to continue the execution. The trade-off made in such a system, thus, is that it gives up on the guarantee of exact replay, in order to gain more flexibility by supporting a range of program modifications.

In this paper, we take a different approach. We aim to provide guaranteed deterministic replay using an RnR log, while allowing programmers to perform the debugging tasks mentioned earlier.

3. Using RnR Support for Debugging

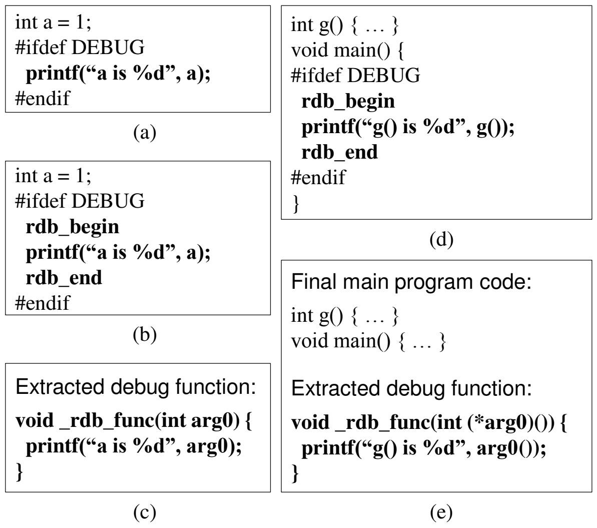
We call *Replay Debugging* the process of debugging a program while replaying its execution using a previously-generated RnR log. In this paper, to quickly diagnose non-deterministic bugs, we are interested in the ability to always *guarantee exact replay* during replay debugging. A requirement for this capability is that the debugging process should not distort the program's code or data in any way. If this requirement is not satisfied, the RnR log becomes obsolete and cannot be used. Unfortunately, many of the features needed in an effective debugging process are at odds with such a requirement. In this section, we discuss *four usability features* that we believe are needed for effective debugging. For each, we outline the challenge it presents to our target environment, and how a system that we propose, called `rdb`, addresses the challenge.

3.1. Inline Debug Code in the Program Code

Programmers typically inline debug code in the program code, as if it were part of the main program. For example, in C/C++ programs, debug code is often enclosed between `#ifdef` and `#endif` pre-processor macros, so that it is included in the compilation of a debug version of the program and is excluded otherwise (Figure 2(a)). This approach enables writing complex debug logic while allowing easy access to the program's state and code.

Challenge. Since the inlined debug code is compiled together with the main program's code, it changes the program's code and data structures, and renders the RnR log obsolete.

Solution. To address this challenge, `rdb` uses a compiler pass to *extract* the debug code from the program code. Programmers can write inlined debug code, but they need to enclose it within special `rdb` markers so that the compiler can identify the enclosed code as debug code. Figure 2(b) shows



(a)
```
int a = 1;
#ifdef DEBUG
  printf("a is %d", a);
#endif
```

(b)
```
int a = 1;
#ifdef DEBUG
  rdb_begin
  printf("a is %d", a);
  rdb_end
#endif
```

(c)
```
Extracted debug function:
void _rdb_func(int arg0) {
  printf("a is %d", arg0);
}
```

(d)
```
int g() { … }
void main() {
#ifdef DEBUG
  rdb_begin
  printf("g() is %d", g());
  rdb_end
#endif
}
```

(e)
```
Final main program code:
int g() { … }
void main() { … }

Extracted debug function:
void _rdb_func(int (*arg0)()) {
  printf("g() is %d", arg0());
}
```

Figure 2: Making debug code work for `rdb`.

the code surrounded by the `rdb_begin` and `rdb_end` macros understood by the compiler.

The step of extracting the debug code should take place in the compiler front-end at the level of the Abstract Syntax Tree (AST) — before any transformation or code generation is done. From this point on, the compiler will compile two different bodies of code: (1) the main program code, which is exactly the same code that was used for generating the binary of the recorded program, and (2) the debug code.

The extracted debug code is transformed before being compiled. This is because it references variables, memory locations and functions of the program that are not available to it after the extraction step. For example, in Figure 2(b), the reference to variable `a` will not be resolvable after the debug code is extracted. Hence, the compiler transforms each group of debug instructions into a *debug function* that receives, as its formal arguments, those variables of the program code that are accessed by the debug code (Figure 2(c)).

In addition, the compiler front-end needs to leave some markers in the main program code to convey to the back-end the location in the program where the debug code should be executed, as well as the variables it will access. The back-end will use these markers to generate extra files with information about the debug functions and their arguments. This information will be used to invoke the extracted debug code.

3.2. Access Program Code & Data from the Debug Code

The debug code needs to be able to read arbitrary variables and memory locations of the main program. It also needs to be able to invoke subroutines in the program — e.g., to evaluate the value of an expression or to traverse program data structures that might in turn call other subroutines. To provide this capability, the debug code should run in the same virtual address space as the main program.

Challenge. Allowing the debug code to use the address space of the program to contain its code and state results in some memory ranges not being available to the main program. If the main program tries to access a location in these ranges, it can result in replay divergence.

Solution. rdb places the debug code and state in those parts of the program address space that are not going to be used by the main program. This is feasible, since the RnR log contains enough information to allow rdb to identify the memory ranges not used by the main program.

3.3. Output the Results of the Debug Code

Inspecting the program's state is not very useful if the inspection results cannot be conveyed back to the developer. For example, in C programs, programmers often use printf.

Challenge. Since the debug code is running in the same address space as the program, it could call the program's printf. However, the call will change the contents of data structures internal to the runtime library (libc in this case) — which are part of the main program state. This change will cause a replay divergence.

Solution. rdb provides the debug code with its own instance of the runtime libraries (e.g., libc and libstdc++ for C/C++ programs). In the code generation phase, the compiler treats calls to such subroutines by the debug code differently than calls to the subroutines of the main program. For example, consider Figure 2(d). The debug code contains two function calls, namely printf() and g(). The compiler identifies printf() as a member of the runtime library and not as an input to the debug code. Later, when the debug code is linked with its own libc, this function will be resolved to the printf in that instance of libc.

On the other hand, function g() comes from the main program. Like any other piece of main program accessed by the debug code, it will be passed to the debug code as an input. Specifically, when the debug function gets called, the location of g() will be passed as an argument to the function. Hence, the debug code calls the program's instance of g(). Figure 2(e) shows the resulting main program and extracted debug code.

3.4. Keep State in the Debug Code

When programmers debug code with complex data structures, they often need to keep some shadow state for debugging purposes. This is usually done by allocating some heap objects that outlive the piece of debug code creating them. They are accessed in the future by some other part of the debug code. In addition, these objects may need to include references to objects belonging to the main program.

Challenge. Debug code cannot allocate its dynamic objects in the same heap as the main program. This would change the program's state and potentially result in replay divergence.

Solution. rdb provides the debug code with its own instance of the runtime library. Hence, the debug code will automatically use the heap that belongs to this runtime library as it invokes memory allocation routines (e.g., malloc()). Recall from Section 3.2 that rdb ensures that the addresses used by the main and debug codes do not interfere. However, since debug code lives in the same virtual address space as the main code, debug objects can easily contain references to objects belonging to the main program.

4. Base Design of Replay Debugging with rdb

We argue that, to guarantee replay debugging with exact replay, we need two capabilities. One is the ability to generate, out of the code instrumented with debug statements, an executable that is identical to that of the original application. The second is the ability to replay the program encoded in the log while invoking the debug code at the appropriate locations in the code. In this section, we describe how rdb attains these two abilities. Before this, we discuss the structure of the debug code. In our discussion, we assume that rdb operates on C/C++ programs.

4.1. Structure of the Debug Code

To replay-debug a program with rdb, a developer writes snippets of debug code inlined in the program code. The inlined debug code should be a single-entry, single-exit region [3] in the control flow graph of the program. This is needed to ensure that the compiler can easily extract the debug code from the program. We call every such piece of debug code a *Debug Region*. Each debug region is enclosed between rdb_begin and rdb_end markers to help the compiler identify it.

The code in a debug region can freely access any object (variable, function or constant) that is accessible by the main program code as long as it does not write, directly or indirectly, to the memory owned by the main program. A debug region can also have locally-declared variables that are only visible in that debug region, and freely use functions provided by its private instance of runtime libraries. Figure 3 shows an example of a debug region with a for loop, a locally-declared variable, and a printf statement.

```
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
}
else {  ... /* program code */ }
```

Figure 3: Example of a debug region.

In addition, the developer can also write new functions to call from the debug region, and declare and use new global variables that do not exist in the original code. These function and global variable declarations are not in a debug region. We explain how rdb supports them in Section 5.1.

4.2. Generating the Executable for Replay Debugging

After the developer has augmented the program source with debug regions, the first step is to generate an executable of

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

(a)

```
@.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()
}
```

(b)

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

void @__rdb_func_1(i8* %arg) {
 %_tmp1 = load %arg
 call @printf(@.str, %_tmp1)
}
```

(c)

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

(d)

Function Descriptors:

FuncID	FuncName
1	__rdb_func_1
2	…

(e)

Argument Descriptors:

FuncID	Position	Class	Info
1	0	Stack	(SP, -20)
2	0	…	…
2	1	…	…

(f)

Figure 4: Compiling an example program for replay debugging: C program containing debug code (a); resulting LLVM IR generated by the Clang front-end (b); extracted debug module (c); resulting main code containing rdb markers (d); function descriptors (e); and argument descriptors (f).

the application that, while identical to the original application in both code and data, can also invoke the debug code. The idea in `rdb` is to force the compilation process to generate two binary files from the program source files. One is identical to the binary of the original program with no debug code; the other encapsulates all the extracted debug code.

To this end, the compiler takes each source file and generates two object files, one with the main program code, and the other with the extracted debug code. After all the files have been processed, the two sets of object files are linked separately to generate two different binaries.

In the following, we describe the operation in detail. We describe it in the context of the Clang/LLVM compilation flow [23], which is outlined in Figure 5. This tool set takes C/C++ source files and, in the front-end (leftmost box), translates them to unoptimized LLVM Intermediate Representation (IR). The output is then taken by the LLVM optimizer (central box), which generates optimized LLVM IR. For simplicity, the current implementation of `rdb` operates under the assumption that the code is compiled without optimization (i.e., with the -O0 command line option). Section 7.2 discusses the extensions needed to handle optimized code. Finally, the output of the central box is taken by LLVM CodeGen backend (rightmost box), which translates it into x86 machine code. `rdb` augments the last two boxes.

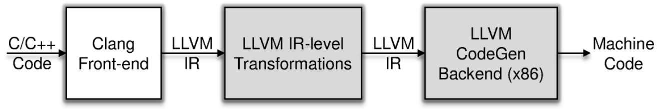


Figure 5: Clang/LLVM compilation flow. rdb augments the two shaded boxes.

To aid the presentation, we use the simple C program in Figure 4(a) as a running example. The original program reads a character from the standard input. The debug code then prints it to the standard output. Figure 6(a) shows the `rdb` compilation flow, which we will describe in steps.

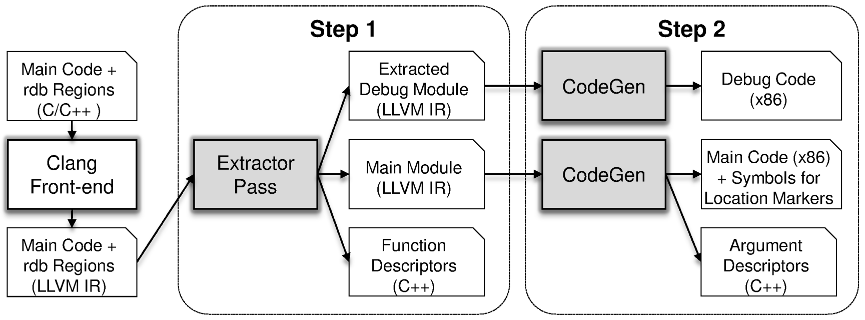
We use the Clang front-end to translate the program source to its equivalent LLVM IR. After translation, the code in a valid debug region retains its shape as a single-entry, single-exit region enclosed between begin and end markers. Figure 4(b) shows the resulting LLVM IR. Following the LLVM convention, names that start with % are virtual registers, while those that start with @ are global objects. For simplicity, we show an abridged version of the LLVM code that, although not complete, captures the essence of the generated IR. `rdb_begin` and `rdb_end` are replaced by calls to two dummy functions that will be removed later. We are now ready to perform the two compilation steps for `rdb`: code extraction and machine code generation.

4.2.1. Step 1: Code Extraction. This step is performed inside the LLVM IR optimizer. It is shown in Step 1 of Figure 6(a). This step, called *Extractor*, extracts the debug code from the input LLVM IR code, and generates two modules. One is the extracted debug code; the other is the resulting main code. The Extractor runs before any further processing of the input LLVM IR code, so that the next compilation steps are guaranteed to operate on the same LLVM IR as in the original code.

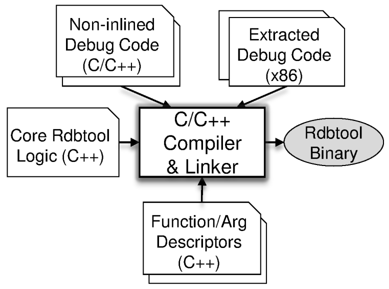
For each debug region, the Extractor generates one debug function, which contains the LLVM code of that region. Any variable or function that belongs to the main code and is accessed in the debug region becomes an argument to the debug function. We call such variables *Debug Arguments*. The Extractor replaces all the references to a debug argument in the body of the debug function with references to the corresponding argument.

Figure 4(c) shows the debug module extracted from the example program. It contains one debug function. The debug region accesses three objects that are not defined in the region: variable `c`, function `printf()`, and constant string `.str`. Variable `c` is an input to the function. The `printf()` function comes from the debug code's `libc`. Finally, `.str` is a constant that would not have existed if it was not used in the debug function. Hence, it should only be part of the debug code. Thus, the single argument of the function is the address of `c` from the main code when the function is invoked.

The resulting main code is the same as the original program code, except for some markers that are added by the Extractor to establish the necessary relation between the main code


Step 1
Step 2
Main Code + rdb Regions (C/C++)
Clang Front-end
Main Code + rdb Regions (LLVM IR)
Extractor Pass
Extracted Debug Module (LLVM IR)
Main Module (LLVM IR)
Function Descriptors (C++)
CodeGen
CodeGen
Debug Code (x86)
Main Code (x86) + Symbols for Location Markers
Argument Descriptors (C++)

(a)


Non-inlined Debug Code (C/C++)
Extracted Debug Code (x86)
Core Rdbtool Logic (C++)
C/C++ Compiler & Linker
Rdbtool Binary
Function/Arg Descriptors (C++)

(b)

Figure 6: `rdb` compilation flow (a), and the different components of the Rdbtool binary (b).

and the debug code. There are two type of markers: (1) *Location* markers, which mark the points in the control flow of the main code where the debug functions should be invoked, and (2) *Argument* markers, which mark the variables that are referenced in the debug region and thus have to be passed as arguments to the corresponding debug function. These markers are represented as LLVM intrinsics, which are calls to built-in functions of the compiler (`llvm.rdb.location()` and `llvm.rdb.arg()`, respectively). They will be processed in Step 2. Figure 4(d) shows the resulting main code, where the whole debug region has been replaced by intrinsic calls.

The arguments of the markers are used to identify the correct debug code. Specifically, each debug region is assigned a unique integer ID by the Extractor. This ID is passed as the first argument to the corresponding location and argument markers in the main code. In Figure 4(d), this is ID 1. To relate these IDs to the debug function names, the Extractor generates a Function Descriptor file that associates an ID to each generated debug function name (Figure 4(e)). Using this information, the replay execution will identify the debug function that has to be invoked at a given marked location. In addition, the argument marker (`llvm.rdb.arg()`) for a variable takes two additional arguments: the position of the variable in the argument list of the debug function, and the variable. In Figure 4(d), the position is 0 because variable `c` is the only argument of the debug function.

4.2.2. Step 2: Machine Code Generation. The second `rdb`-specific compilation step is performed in the CodeGen pass. It is shown in Step 2 of Figure 6(a). This step takes the extracted debug and main modules and translates them to machine code. The debug module does not need any special treatment from CodeGen, since it is normal LLVM code. The main module, however, contains the markers that need to be handled. In this step, we need to ensure that the markers do not change the code generation process relative to the original code. It is at this step that the location of the debug arguments in the main code is determined. Generating a location that is different from a variable's location in the original code will result in an inconsistent execution during replay.

CodeGen removes the argument markers early on — before any code generation activity such as instruction selection or register allocation takes place. In this manner, `rdb` can guarantee that the machine code generated is the same as for the original code. During the code generation, however, these debug variables are tracked, such that we can know what location has been assigned to each of them. After the machine code is finalized, CodeGen outputs an Argument Descriptor file, which has a descriptor for each debug argument. The descriptor for an argument includes the ID of the function to which the argument belongs, the position of the variable in the argument list of that function, and some information about the class of the variable. The latter allows the replay execution to find the location of the variable in the main program when invoking the debug function.

There are three classes of variables that CodeGen tracks: (1) register-allocated variables, (2) stack-allocated variables, and (3) global variables or functions. For register-allocated variables, the descriptor contains the register name. Stack-allocated variables are described by a (*register*, *offset*) pair; *register* is usually one of the stack pointer or frame pointer registers, and *offset* is an immediate value to add to *register*. Global variables and functions are described as a (*symbol*, *offset*) pair. The desired location is calculated by adding *offset* to the location of *symbol* in the address space; the latter is found by looking up *symbol* in the symbol table of the program.

Figure 4(f) shows the argument descriptor file for the example. The first row corresponds to variable `c`. It belongs to function `__rdb_func_1` (ID is 1), it is the function's first argument (Position is 0), and it is found in the stack at offset -20 from the stack pointer (Info is (SP,-20)).

Finally, location markers, which indicate main-code locations at which debug functions should be invoked, are translated to labels in the code. These labels do not affect the code generation process in any way. At the end, they become symbols in the symbol table of the generated machine code. The name of the symbol contains the ID of the corresponding debug function as a suffix. This way, the replay execution knows which debug function to call when the execution flow reaches that location.

4.3. Executing the Debug Code while Replaying

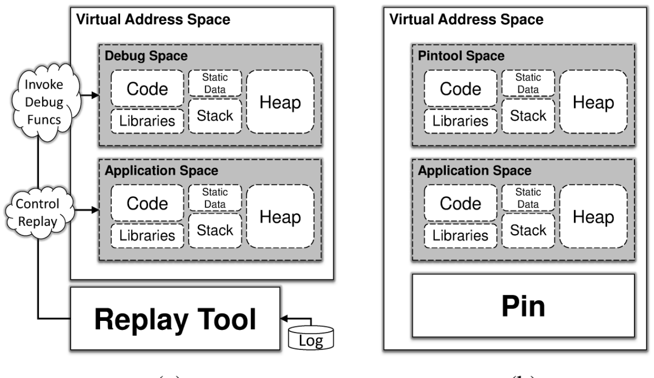
After `rdb` has generated the main and debug binary modules described above, the second mechanism needed for replay debugging is the ability to replay the execution encoded in the logs while invoking the debug code at the appropriate locations in the code. For this, we need an infrastructure with three functionalities.

First, we need to set up a virtual address space that is shared by the main program and the debug code. However, each of the two needs to have its own instance of the runtime libraries, and use different memory ranges for their code and data (stack, static data, and heap).

Second, the infrastructure needs to replay the application using the recorded input and memory access interleaving logs, injecting inputs and enforcing access interleavings as recorded.

Finally, it should provide the ability to invoke the appropriate debug function with appropriate arguments, without affecting the deterministic replay, when the execution flow of the application reaches a marked location. The required steps involve pausing the replay, setting up a stack frame for the debug function without affecting the main program, transferring the control to the debug code (i.e., invoking the function), and returning the control back to the main code when the debug function completes.

Figure 7(a) shows a high-level view of the infrastructure. It contains a replay tool that reads a log, controls application execution, and invokes the debug functions. In this paper, we build the infrastructure using Pin [29]. Specifically, we augment the Pin-based replay mechanism used in QuickRec [36]. The reason is that, as we discuss next, Pin already provides some of the features needed. In Section 5.6, we discuss an alternative replay infrastructure.



(a) (b)

Figure 7: High-level view of the infrastructure for executing the debug code while replaying (a), and address space of an application running under Pin (b).

4.3.1. Replay Debugging Using Pin: Rdbtool. Pin provides much of the required functionality described. The address space of an application that runs under Pin consists of three parts (Figure 7(b)): (1) the application, (2) the Pin infrastructure, and (3) a Pintool, which is a shared library. The Pintool can use Pin's API to monitor and control the execution of the application. Internally, Pin uses binary instrumentation to implement this. When Pin is invoked, it loads the Pintool and provides it with a copy of the runtime libraries `libc` and `stdlibc++`. Then, it lets the Pintool analyze the instructions in the application's code and instrument them. In QuickRec [36], this Pintool provides *replay* functionality. In `rdb`, we further extend it to provide *replay debugging* functionality, and call it *Rdbtool*.

To replay, we need the application binary, the Rdbtool binary, the libraries of both binaries, and the RnR input and memory logs. The memory log is only required for RnR of multi-threaded workloads, and is in the form of a set of totally-ordered chunks. Prior to starting Pin, we analyze the input log to identify all the memory ranges that are going to be used by the application. This can be done by examining the input log entries for `mmap()` and `brk()` system calls. We then make sure that Pin and the Rdbtool do not use these ranges, to ensure correct replay.

The Rdbtool keeps the debug code and data, and will ensure that the debug code executes when needed. As shown in Figure 6(b), the Rdbtool binary is built by compiling together: (1) the code of the core Rdbtool logic (i.e., baseline replay functionality as in QuickRec [36], plus the invocation of debug functions when execution reaches debug markers), (2) the object files with the extracted debug functions, (3) files with other, non-inlined debug code (explained in Section 5.1), and (4) the function and argument descriptor files generated by the modified compiler.

The Rdbtool controls the RnR input and memory logs during replay. To inject application inputs, the Rdbtool instruments system calls, so that it can emulate their results according to the RnR input log. Most system calls are emulated by injecting their results into the application. Some system calls, however, need to be re-executed to create the appropriate kernel-level state for the application — e.g., memory mapping and thread creation system calls. As for the RnR memory access interleaving log, as chunks replay, a counter counts the instructions executed. When the counter reaches the logged chunk size, the thread's execution is paused and it looks for the next chunk in the log to execute.

Most importantly, the Rdbtool manages the replay debugging. When the Rdbtool is loaded by Pin, it first searches the symbol table of the main program for symbols that mark code locations at which debug functions should be called. When it instruments the application's binary, it instruments these code locations to set *breakpoints*. When execution hits one of these breakpoints, the Rdbtool pauses the replay, and uses the information in the descriptor files to find the address and arguments of the corresponding debug function to call. Then, it calls the function. Note that this function call takes place on the Rdbtool's stack, rather than on the application's stack, to avoid changing the application's memory. Once the debug function completes, the Rdbtool transfers execution to the main program.

5. Advanced Issues

We now describe several advanced issues in the `rdb` design. The last two are discussed here for completeness but have not been implemented in the current system.

5.1. Debug-Only Functions and Global Variables

In the process of debugging, developers often need to define global objects (variables or functions) for use in the debug code. The definitions of such objects can only be included in the debug binary; including them in the main binary would result in a program that is different from the original program. To ensure this, `rdb` requires that the developer writes the definitions of such global objects in source files that are linked with the extracted debug code, to form the Rdbtool binary. Such files are shown as the box labeled Non-inlined Debug Code in Figure 6(b).

When a global object is accessed in a debug region, the Extractor pass needs to know whether it belongs to the main code or it is a debug-only object. A reference to a debug-only global object is not changed, and is resolved at link time when the support file containing the definition of the object is linked-in. A reference to a global object belonging to the main code is turned into a debug function argument.

In our current implementation, the Extractor makes the decision based on the name of the object. All debug-only global object names are required to have a particular prefix. A more elegant solution would involve using C/C++ attributes for this purpose — e.g., each debug-only global object could be marked with a C/C++ attribute named `rdb` to make it easy to identify.

5.2. Event-Driven Debugging

Developers often like to invoke debug code when a certain event happens in the main application — rather than when execution reaches a marked location. This is called event-driven debugging, and is supported in `rdb` with a certain API. Developers can use this API to associate call-back functions with events, rather than marking the application code. The Rdbtool then adds instrumentation to the application code to detect the occurrence of the events. When an event happens, the associated call-back function is invoked.

There are several events that the developer can ask `rdb` to monitor. One is the occurrence of a system call. The associated call-back is invoked before or after a system call executes. For example, in some programs, buffer overflow or under-synchronized buffer accesses can result in gibberish program output. By asking `rdb` to monitor `write()` system calls, one can identify the code responsible for the bug.

Another event is the call of a function. The associated call-back is invoked when an arbitrary function is called. This is especially useful to monitor library calls in a program.

Finally, another event is reading or writing a certain memory location. This corresponds to the popular "watchpoint" functionality. In this case, the associated call-back function is invoked before or after the program execution accesses the location. This functionality is useful to diagnose bugs such as segmentation faults or buffer overflows. Currently, this functionality is implemented in `rdb` by monitoring each memory access, and comparing the accessed address to the watched address, and invoking the call-back function if they match. A future implementation will involve using the watchpoint registers provided by the x86 processor hardware.

5.3. Protecting Against Writes to Main-Program Memory

The debug code should not write directly or indirectly to memory regions of the main program. To enforce this, the Rdbtool can optionally change the access protection of main-program memory regions to read-only prior to invoking a debug function. It then restores the original protections after executing the debug code. This comes at a performance cost, but detects debug code that violates the read-only-access requirement.

5.4. Using gdb with Replay Debugging

Pin can be connected to gdb, giving gdb full control over the execution of the application running under Pin. This way, gdb can be used to debug the application as if the debugger was directly attached to the application. This feature is independent of `rdb`, so it is possible to use gdb even during replay. However, only a subset of gdb features are safe to use in this fashion — namely, those that do not modify the application's memory content (code or data), such as reading the application's memory or setting breakpoints.

More complex debugging logic has to be implemented as `rdb` debug code, to avoid affecting the application state. Examples of such debug logic include adding local, global and dynamic objects in the debug code, adding and executing new code, including if statements, while loops, and function calls/definitions, or creating shadow data structures. The support of such complex debug code is one of the main features that distinguishes `rdb` from merely using gdb in conjunction with a replay tool (e.g., gdb plus QuickRec).

Still, in an `rdb`-based debugging scenario, using gdb can be particularly useful for debugging tasks that are not easy to do using inlined debug code, such as back-tracing an application's stack or single-stepping through the execution.

5.5. Replay Debugging with Partial Logs

In long-running recordings, the recorded log size can grow very large. To reduce storage requirements, periodic snapshots of the application state could be taken. In this case, when a snapshot is taken, the recorded log up to that point would be purged. Thus, in this environment, the execution would be recorded as an application snapshot plus a partial log that records the rest of the execution.

For `rdb` to perform replay debugging in such an environment, it would first have to initialize the state of the application using the snapshot; then it could replay the events in the partial log. Since the program being replayed is exactly the same (in terms of both code and data content) as the recorded program, `rdb` would work correctly after restoring the snapshot.

5.6. Replay Debugging without Pin

Section 4.3.1 described how `rdb` uses Pin to support the second mechanism needed for replay debugging: executing the debug code while replaying the main program. In reality, `rdb` can be built on top of other replay infrastructures. One of them is replay using OS functionality, as exemplified by Cyrus [17] and SCRIBE [22].

In this case, the role of the Replay Tool in Figure 7(a) is played by a modified OS kernel. The OS creates a process, loads into memory the code of the application to be replayed and the libraries it uses, and then starts replaying the application. The OS injects the recorded inputs from the input log (e.g., when the program makes system calls), and enforces the memory access interleavings from the memory log, using mechanisms explained in Cyrus [17]) and SCRIBE [22].

To support replay debugging in this environment, the OS also needs to load the binary for the debug code, and link it with a separate instance of the run-time library. The OS can easily make sure that the application and the debug-code binaries use distinct address ranges. To mark debug locations, the OS can use either hardware or software breakpoints. When a breakpoint is hit, control transfers to the OS. The OS can then calculate the address and arguments of the corresponding debug function using the information in the descriptor files. Then, it sets up a dummy stack in an unused part of the address space, sets the program's PC to point to the first instruction of the debug function, and transfers control back to user mode. In this way, when the program resumes execution, it will execute the debug function. After the function terminates, the OS transfers control back to the main program.

This technique to invoke user-mode code by the kernel is the same mechanism used in Linux, for example, to invoke signal handlers. These handlers are functions defined in user-mode code that are executed when the kernel receives signals destined for the process.

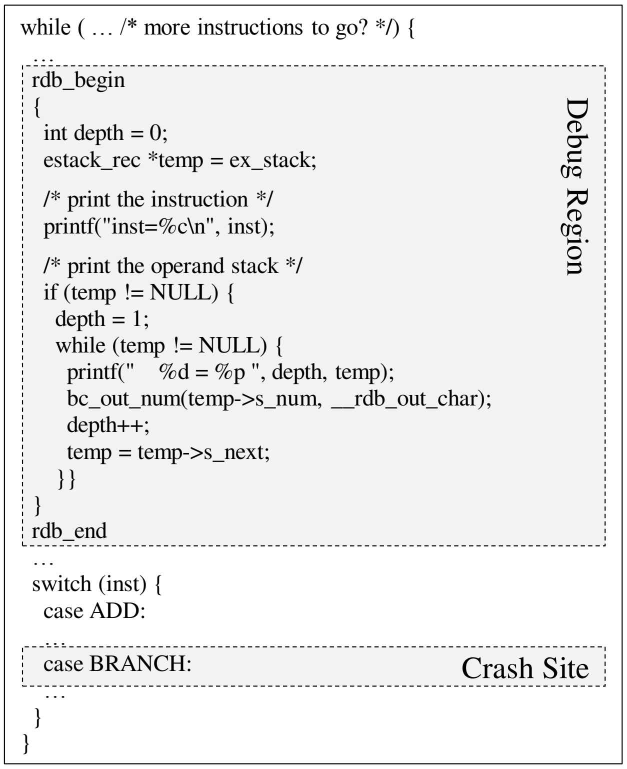
6. An Example of Replay Debugging

To illustrate replay debugging with `rdb`, we examine a bug in the GNU *bc* program version 1.06 [14] that crashes the program due to a segmentation fault [15]. This bug is also included in the BugBench bug-benchmark suite [28]. While this bug is not timing-dependent or multithreaded, we examine it because it illustrates many of `rdb`'s capabilities.

bc is a popular numeric processing program that takes as input a program with C-like syntax and executes it. *bc* works by first translating its input program to an internal byte-code format (translation phase) and then executing the byte-code (execution phase). In this section, "instruction" refers to a byte-code instruction. Instructions read their operands from an "operand stack" and push the result back on the stack.

We assume that a user was running the *bc* program on a machine equipped with RnR hardware when the crash happened, and that he/she gave us the resulting RnR log. We now consider the replay debugging process in steps.

Step 1: Replay to Find Out the Crash Point. The first step is to find out where the crash happens in the program code. For this, we attach gdb to the replayer (Section 5.4) and replay the execution. When the program crashes, we use gdb to analyze its stack frames at the crash point. We conclude that the crash happens while executing the byte-code (the execution phase). Specifically, it happens in the BRANCH case of a switch statement inside a while loop that processes each instruction *inst* that gets executed (Figure 8(a)).



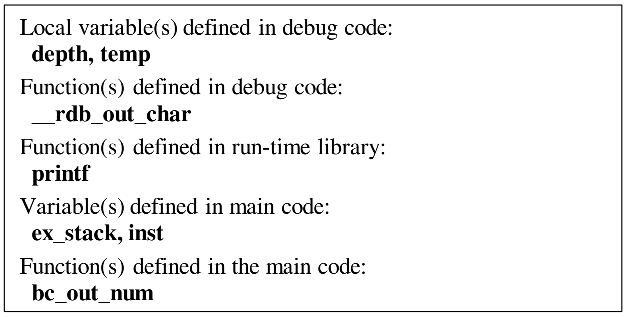
```
while ( … /* more instructions to go? */) {
 …
 rdb_begin
 {
  int depth = 0;
  estack_rec *temp = ex_stack;

  /* print the instruction */
  printf("inst=%c\n", inst);

  /* print the operand stack */
  if (temp != NULL) {
    depth = 1;
    while (temp != NULL) {
      printf("    %d = %p ", depth, temp);
      bc_out_num(temp->s_num, __rdb_out_char);
      depth++;
      temp = temp->s_next;
    }}
 }
 rdb_end
 …
 switch (inst) {
  case ADD:
  …
  case BRANCH:
  …
 }
}
```

Debug Region
Crash Site

(a)



Local variable(s) defined in debug code:
depth, temp
Function(s) defined in debug code:
__rdb_out_char
Function(s) defined in run-time library:
printf
Variable(s) defined in main code:
ex_stack, inst
Function(s) defined in the main code:
bc_out_num

(b)

Figure 8: Example using `rdb` for replay debugging: program with a debug region (a) and objects accessed in the debug region (b).

Step 2: Replay Again to Print State at the Crash Point. The next step is to find out why the crash happens. For this, we write a debug region before the crash point that prints *inst* and the contents of the operand stack when *inst* executes. Figure 8(a) shows the debug code inside the `rdb_begin` and `rdb_end` markers. Figure 8(b) shows the objects accessed in the debug code. They include local variables defined in the debug code (`depth` and `temp`), functions defined in the debug code (`__rdb_out_char`, but the definition is not shown), functions from the run-time library of the debug code (`printf`), variables defined in the main code (`ex_stack` and `inst`), and functions from the main code (`bc_out_num`).

The main program includes the `bc_out_num` function that is used to write numbers to the output. Internally, it first computes the characters that need to be put out and then, instead of directly writing them to the output, it passes each character to a pretty-printing function which is passed to `bc_out_num` as an argument. This second function does the actual output. In the debug code, when we call `bc_out_num`, we cannot pass to it any of the pretty-printing functions defined in the main program, since they eventually call functions from the `libc` of the main code. This would result in replay failure. Instead, we define an equivalent function in the debug code, called `__rdb_out_char`, and pass it as an argument to `bc_out_num`.

Step 3: Identify that the Problem Is Somewhere Else. Based on the data printed by the debug code, we find that the program crashes because a BRANCH finds an empty operand stack while it expects to find the branch condition on the stack. Now we know that the actual problem is in the translation phase — the instructions preceding the BRANCH do not produce correct operand stack state. Consequently, we need to examine the input program of *bc* and find the portion of it that generates the instructions before the branch.

Step 4: Replay Again to Print State at the New Point. To obtain the input code that generates the instructions before the branch, we add a new debug region in the translation code. The region prints the input program from *bc*'s internal data structures before it is translated.[1] After this, we replay the program again and print the code.

Step 5: Diagnose the Bug. We compare the output of Step 2 (instructions and stack content) to the output of Step 4 (input program) to find the bug. The input program contains a `for` loop whose condition is empty. This is equivalent to a `true` condition, which means that the body of the loop should be executed. Unfortunately, for this pattern, the buggy translator fails to include an instruction that pushes the constant `true` on the stack, and the subsequent branch instruction crashes.

This example has shown several `rdb` features, such as: (1) the combined use of gdb and `rdb`, (2) three deterministic replays of the program with expanding debug instrumentation, and (3) debug regions that use many different types of objects. The replays would be deterministic even for timing-dependent, multithreaded bugs.

7. Current Limitations and Potential Solutions

We now discuss the main limitations of the current `rdb` design.

7.1. Adding/Removing Code in the Main Program

Since `rdb` targets replay debugging with guaranteed exact replay, it cannot tolerate changes to the main program that are not extracted into the debug code binary. In a debugging process, after `rdb` has helped diagnose the bug, the code will be patched to fix the defect. Patching involves adding and/or removing code in the program. After the code is patched, `rdb` cannot be used for replay debugging the resulting program using the original log. This is a fundamental limitation of replay debugging with guaranteed exact replay.

[1]Alternatively, we could ask the user who gave us the log to also provide the input program. However, we can easily regenerate it ourselves.

7.2. Supporting Compiler Optimizations

A limitation of the current implementation of `rdb` is the assumption that compiler optimizations have been disabled. Disabling optimizations results in that the generated LLVM IR and machine code are in direct correspondence with the high-level program. This makes the debug code extraction and code generation processes of Section 4 easier to implement.

The difficulty with compiler optimizations is that, because they are applied *after* the debug code has been extracted from the program, the compiler performs them without being aware of the debug code. Hence, the compiler may optimize away some of the state that the debug code will attempt to access. In general, the compiler may perform optimizations that are invalid in the presence of the debug code.

Figure 9(a) shows an example. In this program, character `c` is read from the input and variable `a` is set (gray box in the figure). However, `a` is not used in the main code — it is only used in the debug code. After we extract the debug region, a Dead Code Elimination (DCE) pass will remove the statement in the gray box from the main code as dead code. The DCE optimization has to be performed because it was also performed in the original program recorded in the log. However, this optimization causes the debug code executed during replay debugging to fail.

```
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
}
```

(a)

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

(b)

Figure 9: Optimization example: program before (a) and after (b) automatic debug code insertion.

To assess the applicability of `rdb` to optimized code, we analyzed the optimizations that LLVM performs in its -O2 and -O3 optimization levels. Based on our observations, we propose a strategy for a future improved version of `rdb` that handles optimized code.

First, we find that some optimizations (e.g., Common Subexpression Elimination (CSE), loop unrolling and inlining) do not affect the validity of the debug code. The reason is that these optimizations do not optimize away the main program state accessed by the debug code. Hence, the compiler can safely repeat such optimizations in the presence of `rdb` markers.

However, there are other optimizations where the compiler has to optimize based on the knowledge of both the main and debug codes. In some cases, the compiler can automatically generate extra debug code after performing an optimization on the main code. This extra debug code *undoes* some of the effects of the optimization for the debug code. As an example, consider the code in Figure 9(b). After having removed the statement in the gray box in Figure 9(a) from the main program, the compiler adds a new debug region (gray box in Figure 9(b)) that calculates `a` in the debug code for later use. With this change, the compiler can optimize the main code exactly the way it optimized the original code, while keeping the debug code valid. Note that, for simplicity, this example presents the changes in C code. In practice, the extra code will be inserted at the level of the LLVM IR or machine code.

Sometimes, it may be hard or even impossible to undo optimization effects in the above fashion — in particular, when complex pointer-aliasing relations exist between the debug code and the main code. In these cases, the compiler can generate an error message to let the programmer know about the problem. Then, the programmer can change the debug code accordingly. Overall, we believe it is possible to extend `rdb` to work with optimized code while supporting replay debugging with guaranteed exact replay.

7.3. Cross-Region Data Sharing

In the current implementation of `rdb`, it is not possible to define a debug-only *local* variable in one debug region and use it in another debug region in the same function. This is because we convert each debug region into a separate function. Thus, all of the cross-region data sharing has to happen through debug-only *global* variables (Section 5.1).

This inconvenience can be easily relieved by adding compiler support for automatically converting such local variables to global variables. Using stack-like data structures, it is also possible to support situations (such as recursive function calls) in which multiple instances of the same static debug-only local variable are simultaneously alive. We leave the details of the design to future work.

8. Related Work

In general, debugging involves bug reproduction, diagnosis and fixing. While there are many proposals for using RnR to reproduce bugs (e.g., [4, 9, 10, 13, 16, 21, 24, 33, 35, 40, 41]), very few have tackled the issues of bug diagnosis and fixing.

Some RnR proposals [9, 20, 35, 44] allow limited execution inspection capabilities. Aftersight [9], IntroVirt [20] and Simics Hindsight [44] allow programmers to write code that inspects the state of the program under replay. They all record and replay virtual machines rather than individual applications. As a result, the kind of inspection code they support is different in nature than debug code that can be inlined with main code. In addition, Aftersight and Hindsight keep the debug state in a separate address space than the program being debugged, and IntroVirt does not allow debug code to keep state. Hence, neither provides all of the usability features mentioned in Section 3. PinPlay [35] uses Pin [29] for both record and replay. Similar to `rdb`, it uses Pin's gdb-connection feature (Section 5.4) to let the debugger control and inspect the application's execution. However, to avoid replay divergence, it can only use the limited set of features explained in Section 5.4.

DORA [43] specifically targets bug diagnosis and patch testing using RnR. Its underlying RnR system, SCRIBE [22], uses a modified Linux kernel to record program inputs as well as inter-thread data dependences. Given the logs recorded by SCRIBE, DORA then uses a search-based algorithm to allow "mutable replay" of modified programs, as explained in Section 2.2. DORA does not guarantee deterministic replay, and hence, cannot ensure exact debug-time reproduction of non-deterministic events that resulted in a bug. This is a major limitation and affects its usability as a replay debugging tool.

In addition, to reduce the recording overhead, SCRIBE's approach to recording memory-access interleavings systematically perturbs a program's shared-memory accesses. Moreover, DORA's replay of memory interleavings in multithreaded programs relies on SCRIBE's particular style of recording them. This design choice negatively affects DORA's usefulness for capturing, reproducing and debugging concurrency-related bugs such as data races and atomicity violations.

There is a vast body of research on hardware-assisted MRR [5, 7, 17, 18, 19, 30, 31, 32, 33, 36, 37, 38, 39, 45, 46, 47]. Recent proposals record the execution of a thread as a sequence of chunks of instructions, and inter-thread dependences as orders between chunks of different threads. While most proposals only consider the problem of capturing memory races, a few also include designs that integrate software support for recording program inputs, to enable application-level RnR [17, 31, 36]. BugNet [33] takes a different MRR approach and records processes by storing the result of load instructions in a hardware-based dictionary. This is enough to handle both input and memory-interleaving non-determinism. Lee et al. [25, 26] augment this technique by using offline symbolic analysis to reconstruct the inter-thread dependences. Overall, none of these hardware-assisted RnR proposals consider the problem of using their RnR system for replay debugging.

Software-only RnR solutions rely on modified runtime libraries, compilers, operating systems and virtual-machine monitors to capture sources of non-determinism [4, 6, 8, 9, 11, 12, 21, 22, 24, 27, 34, 35, 40, 41, 42]. Some record and replay execution of individual applications [4, 8, 22, 24, 27, 34, 35, 40, 41, 42], while others operate at the level of virtual machines [6, 9, 11, 12, 21]. All RnR solutions that provide debugging capability beyond bug reproduction (discussed at the beginning of this section) use software-only RnR.

9. Concluding Remarks

While hardware-assisted RnR has been proposed as a primitive for debugging hard-to-repeat software bugs, simply providing support for repeatedly stumbling on the same bug does not help diagnose it. For bug diagnosis, developers need to modify the code — e.g., by creating and operating on new variables or printing state. Unfortunately, this renders the RnR log inconsistent.

This paper introduced `rdb`, the first scheme for replay debugging that guarantees exact replay. With `rdb`, the user interface is the same as in an ordinary bug diagnosis session: the user can read program variables, invoke program functions, create new variables and functions, set watchpoints, and print state. `rdb` uses the log generated by hardware-assisted RnR to always guarantee deterministic re-execution. `rdb`'s operation is possible thanks to two mechanisms. The first one is a compiler mechanism that splits the instrumented application into two binaries: one that is identical to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime one that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. This paper described an implementation of `rdb` using LLVM and Pin, and discussed an example of how `rdb`'s replay debugging is used to diagnose a real bug.

References

[1] "GDB: The GNU Project Debugger," http://www.gnu.org/software/gdb/.

[2] "The LLVM Compiler Infrastructure," http://llvm.org/.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Addison-Wesley, 2007.

[4] G. Altekar and I. Stoica, "ODR: Output-Deterministic Replay for Multicore Debugging," in *SOSP*, October 2009.

[5] A. Basu, J. Bobba, and M. D. Hill, "Karma: Scalable Deterministic Record-Replay," in *ICS*, June 2011.

[6] T. Bressoud and F. Schneider, "Hypervisor-Based Fault-Tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, February 1996.

[7] Y. Chen, W. Hu, T. Chen, and R. Wu, "LReplay: A Pending Period Based Deterministic Replay Scheme," in *ISCA*, June 2010.

[8] J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," in *SPDT*, August 1998.

[9] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling Dynamic Program Analysis from Execution in Virtual Environments," in *USENIX ATC*, June 2008.

[10] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen, "Multi-Stage Replay with Crosscut," in *VEE*, March 2010.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," in *OSDI*, December 2002.

[12] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution Replay of Multiprocessor Virtual Machines," in *VEE*, March 2008.

[13] S. I. Feldman and C. B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," in *PADD*, May 1988.

[14] Free Software Foundation, "bc - GNU Project," http://www.gnu.org/software/bc.

[15] ——, "Bug in GNU bc-1.06," http://lists.gnu.org/archive/html/bug-gnu-utils/2001-02/msg00118.html.

[16] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An Application-level Kernel for Record and Replay," in *OSDI*, December 2008.

[17] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira, "Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism," in *ASPLOS*, March 2013.

[18] N. Honarmand and J. Torrellas, "RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors," in *ASPLOS*, March 2014.

[19] D. R. Hower and M. D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," in *ISCA*, June 2008.

[20] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates," in *SOSP*, October 2005.

[21] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," in *USENIX ATC*, April 2005.

[22] O. Laadan, N. Viennot, and J. Nieh, "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems," in *ACM SIGMETRICS*, June 2010.

[23] C. Lattner, "The Architecture of Open Source Applications, chapter LLVM," http://www.aosabook.org/en/llvm.html.

[24] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Comp.*, April 1987.

[25] D. Lee, M. Said, S. Narayanasamy, and Z. Yang, "Offline Symbolic Analysis to Infer Total Store Order," in *HPCA*, February 2011.

[26] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira, "Offline Symbolic Analysis for Multi-Processor Execution Replay," in *MICRO*, December 2009.

[27] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism," in *ASPLOS*, March 2010.

[28] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for Evaluating Bug Detection Tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[30] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *ISCA*, June 2008.

[31] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay," in *ASPLOS*, March 2009.

[32] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," in *ASPLOS*, October 2006.

[33] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," in *ISCA*, June 2005.

[34] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors," in *SOSP*, October 2009.

[35] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," in *CGO*, April 2010.

[36] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas, "QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs," in *ISCA*, June 2013.

[37] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai, "Architecting a Chunk-Based Memory Race Recorder in Modern CMPs," in *MICRO*, December 2009.

[38] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, H. Jungwoo, and Y. Wu, "CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors," in *MICRO*, December 2011.

[39] X. Qian, H. Huang, B. Sahelices, and D. Qian, "Rainbow: Efficient Memory Dependence Recording with High Replay Parallelism for Relaxed Memory Model," in *HPCA*, February 2013.

[40] Y. Saito, "Jockey: A User-space Library for Record-replay Debugging," in *AADEBUG*, September 2005.

[41] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *USENIX ATC*, June 2004.

[42] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "DoublePlay: Parallelizing Sequential Logging and Replay," in *ASPLOS*, March 2011.

[43] N. Viennot, S. Nair, and J. Nieh, "Transparent Mutable Replay for Multicore Debugging and Patch Validation," in *ASPLOS*, March 2013.

[44] Virtutech, "Using Simics Hindsight for Software Development," http://www.virtutech.com/files/manuals/using-simics-for-software-development_0.pdf.

[45] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar, "Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording," in *ISCA*, June 2010.

[46] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *ISCA*, June 2003.

[47] ——, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *ASPLOS*, 2006.