R2: An Application-Level Kernel for Record and Replay

Zhenyu Guo[†] Xi Wang[‡] Jian Tang[†] Xuezheng Liu[†]
Zhilei Xu[‡] Ming Wu[†] M. Frans Kaashoek[§] Zheng Zhang[†]
[†]*Microsoft Research Asia* [‡]*Tsinghua University* [§]*MIT CSAIL*

ABSTRACT

Library-based record and replay tools aim to reproduce an application's execution by recording the results of selected functions in a log and during replay returning the results from the log rather than executing the functions. These tools must ensure that a replay run is identical to the record run. The challenge in doing so is that only invocations of a function by the application should be recorded, recording the side effects of a function call can be difficult, and not executing function calls during replay, multithreading, and the presence of the tool may change the application's behavior from recording to replay. These problems have limited the use of such tools.

R2 allows developers to choose functions that can be recorded and replayed correctly. Developers annotate the chosen functions with simple keywords so that R2 can handle calls with side effects and multithreading. R2 generates code for record and replay from templates, allowing developers to avoid implementing stubs for hundreds of functions manually. To track whether an invocation is on behalf of the application or the implementation of a selected function, R2 maintains a mode bit, which stubs save and restore.

We have implemented R2 on Windows and annotated large parts (1,300 functions) of the Win32 API, and two higher-level interfaces (MPI and SQLite). R2 can replay multithreaded web and database servers that previous library-based tools cannot replay. By allowing developers to choose high-level interfaces, R2 can also keep recording overhead small; experiments show that its recording overhead for Apache is approximately 10%, that recording and replaying at the SQLite interface can reduce the log size up to 99% (compared to doing so at the Win32 API), and that using optimization annotations for BitTorrent and MPI applications achieves log size reduction ranging from 13.7% to 99.4%.

1 INTRODUCTION

Replay is a powerful technique for debugging applications. When an application is running, a record and replay tool records all interactions between the application and its environment (e.g., reading input from a file, receiving a message). Then when a developer wants to track down an error, she can replay the application to a given state based on the recorded interactions, and investigate how the application reached that state. Replay is particularly useful in the context of distributed applications with many processes. If one of the processes has an error, the developer can investigate the error by replaying that single process instead of all processes, observing the external interactions in the same order as during the recording.

R2 is a novel record and replay tool that allows developers to choose at what interface the interactions between the application and its environment are recorded and replayed. R2 resides in the application's address space and intercepts all functions in the chosen interface. R2 uses a library-based approach to simplify deployment, compared to hardware or virtual machine based approaches. During recording, R2 executes the intercepted calls and records their results in a log. During replay, the application runs as usual (i.e., making library and systems calls, modifying memory, etc.), but R2 intercepts calls in the chosen interface, prevents the real implementation of the calls from executing, and instead gives the application the results of the calls that were previously recorded in the log.

R2 allows developers to choose the interposed interface for two reasons: correctness and performance. The developers can choose an interface that is easy to make *replay faithful*. If an interface is replay faithful, the replay run of an application is identical to the recorded run of the application. This property ensures that if a problem appears while the application is running in recording mode, the problem will also appear during replay. (R2 does *not* attempt to make the behavior of an application with or without recording identical. That is, if a problem appears in the application while it is not being recorded, R2 does not guarantee that the problem will happen again when the application is recorded.)

Achieving faithful replay can be challenging because only calls on behalf of the application should be recorded, recording the effects of a call can be difficult, and not executing calls during replay, multithreading, the presence of the tool in the application's address space may cause the application to behave differently during replay. Previous library-based tools (e.g., liblog [10] and Jockey [28]) interpose a fixed low-level interface and omit calls that are difficult to make replay faithful, and thus limit the applications they can replay.

Consider recording and replaying at the system call interface, which is a natural choice because the application interacts with its environment through system calls. It is not easy to record the output of all system calls. For

Annotation	Scope	Description	Section
in	parameter	input (read-only) parameter	§ 3
out	parameter	output (mutable) parameter	§ 3
bsize(*val*)	parameter	modified size of an array buffer (*val* can be any expression)	§ 3
xpointer(*kind*)	parameter	address allocated internally (*kind* can be *null*, *thread*, or *process*)	§ 3
prepare(*key*,*buf*)	function	prepare asynchronous data transfer	§ 3
commit(*key*,*size*)	function	commit asynchronous data transfer	§ 3
callback	parameter	callback function pointer (upcall)	§ 4
sync(*key*)	function	causality among syscalls and upcalls (*key* can be any expression)	§ 4
cache	function	cache for reducing log size	§ 6
reproduce	function	reproduce I/O for reducing log size	§ 6

Table 1: Annotation keywords (for data transfer, execution order, and optimization).

example, to ensure faithful replay the developer must arrange to record the results of `socketcall` correctly but its results vary for different parameters. For such cases R2 makes it easy for a developer to choose an interface consisting of higher-level functions that cause the same interactions with the environment, but are easier to record and replay. For example, the developer may choose `recv`, which calls `socketcall`; `recv`'s effects are easier to record and replay.

The second reason for allowing developers to choose the interface is that they can choose an interface that results in low recording overhead for their applications. Low overhead is important because the developers can then run their applications in recording mode even during deployment, which may help in debugging problems that show up rarely. To reduce overhead, a developer might choose to record and replay the interactions at a high-level interface (e.g., MPI and SQL library interface such as SQLite) because less information must be recorded. In addition, these higher-level interfaces may be easier to be replay faithful.

To lower the implementation effort for intercepting, recording, and replaying a chosen interface, R2 generates stubs for the calls in the chosen interface and arranges that these stubs are called when the application invokes the calls. The stubs perform the recording and the replay of the calls. To ensure that these stubs behave in way that is replay faithful, the developer must annotate the interface with simple annotations (see Table 1) that specify, for example, how data is transferred across the interposed interface for calls that change memory in addition to having a return value. To reduce the effort of annotating R2 reuses existing annotations from SAL [13] for Windows API. Inspired by the kernel/user division in operating systems, R2 uses a mode bit, which stubs save and restore, to track if a call is on behalf of the application and should be recorded.

We have implemented R2 on Windows, and used it to record and replay at three interfaces (Win32, MPI, and SQLite API). It has successfully replayed various system applications (see Section 8), including applications that cannot be replayed with previous library-based tools. R2 has also replayed and helped to debug two distributed systems, and has been used as a building block in other tools [20, 31, 22].

The main contributions of the paper are: first, a record and replay tool that allows developers to decide which interface to record and replay; second, a set of annotations that allows strict separation of the application above the interposed interface and the implementation below the interface, and that reduces the manual work that a developer must do; third, an implementation of a record and replay library for Windows, which is capable of replaying challenging system applications with low recording overhead.

The rest of the paper is organized as follows. Section 2 gives an overview of the design. Section 3 and 4 describe the annotations for data transfers and execution orders, respectively. Section 5 discuss how to record and replay the MPI and SQLite interfaces. Section 6 and 7 describe annotations for optimizations and implementation details, respectively. We evaluate R2 in Section 8, discuss related work in Section 9, and conclude in Section 10.

2 Design Overview

A goal of R2 is to replay applications faithfully. To do so the calls to intercept must be carefully chosen and stubs must handle several challenges. This section starts with an example to illustrate the challenges, and then describes how R2 addresses them.

2.1 An Example and Challenges

Faithful replay is particularly challenging for system applications, which interact with the operating system in complicated ways. Consider Figure 1, a typical network program on Windows: a thread binds a socket to an I/O port (`CreateIoCompletionPort`,

```
1  struct iocb {
2      OVERLAPPED ov;
3      void * buf, * user_data;
4  };
5
6  int main() {
7      HANDLE hPort = ...;
8      for (...)
9        CreateThread(..., WorkerThread, hPort, ...);
10     ...
11     SOCKET s = socket(...);
12     CreateIoCompletionPort(s, hPort, ...);
13     struct iocb * cb = (struct iocb *)malloc(...);
14     cb->buf = malloc(BUFSIZ);
15     cb->user_data = ...;
16     BOOL fSucc = ReadFileEx(s, cb->buf, BUFSIZ,
17                             (OVERLAPPED *)&cb, 0);
18     ...
19 }
20
21 DWORD WINAPI WorkerThread(HANDLE hPort) {
22     for ( ; ; ) {
23        struct iocb * cb;
24        DWORD size;
25        GetQueuedCompletionStatus(hPort, &size, ...,
26                                  (OVERLAPPED *)&cb, ...);
27        void * buf = cb->buf;
28        void * user_data = cb->user_data;
29        ...
30     }
31     return 0;
32 }
```

Figure 1: A typical network program using asynchronous I/O and completion port on Windows. The pattern is also widely available on other platforms, such as Linux aio (`io_getevents` etc.), Solaris event completion (`port_get` etc.), and FreeBSD kqueue.

line 12), enqueues an asynchronous I/O request (`ReadFileEx`, line 16), and a worker thread waits on the I/O port for the completion of the I/O request (`GetQueuedCompletionStatus`, line 25). Similar interfaces are provided on other operating systems such as Linux (`aio`), Solaris (`port`), and BSD (`kqueue`), and are used by popular software such as the lighttpd web server that powers YouTube and Wikipedia.

The first challenge a developer must address is what calls are part of the interface that will be recorded and replayed. For example, in Figure 1, a developer might choose `socket` but not `ReadFileEx`. However, since during replay the call to `socket` is not executed, the returned socket descriptor is simply read from the recorded log rather than created. So the choice may crash `ReadFileEx` during replay and fail the application; the developer should choose both functions, or a lower layer that `ReadFileEx` uses. Section 2.2 formulates a number of rules that can guide the developer.

R2 generates stubs for the functions that the developer chooses to record and replay, and arranges that invocations to these functions will be directed to the corresponding stubs. To avoid reimplementing or modifying the implementation of the interposed interface, R2's goal is for the stubs to call the original intercepted functions and to record their results. This approach also allows R2 to record and replay functions for which only the binary versions are available.

To achieve this implementation goal and to ensure faithful replay, the stubs must address a number of implementation challenges. Consider the case in which the developer selects the functions from the Windows API (e.g., `GetQueuedCompletionStatus`, `ReadFileEx`, etc.) as the interface to be interposed. During a record run, the stubs must record in a log the socket descriptor and the completion port as integers, the output of `GetQueuedCompletionStatus` (e.g., the value of `cb` at line 26 and the content of `cb->buf` at line 27), along with other necessary information, such as the timestamp when the operating system starts `WorkerThread` as an upcall (callback) via a new thread.

During a replay run, the stubs will not invoke the intercepted functions such as `ReadFileEx` or `GetQueuedCompletionStatus`, but instead will read the results such as descriptors, the value of `cb`, and the content for `cb->buf` from the log. The stubs must also cause the memory side effects to happen (e.g., copying content into `cb->buf`). Finally, the replay run must also deliver upcalls (e.g., `WorkerThread`) at the recorded timestamps.

These requirements raise the following challenges:

- Use of intercepted functions by the implementation of the interposed interface. For example, the implementation itself may invoke the function `socket` and those invocations should not be recorded.

- Functions that have side effects. For example, to record and replay `ReadFileEx`, the stubs must record the content of `cb->buf` and fill it during replay. The stub for `ReadFileEx` must know that the second argument has side effects.

- Addresses returned by `malloc` must be identical during recording and replay. The code in Figure 1 requires that the value that `cb` receives at line 13 must *not* change from record to replay: the replay run reads the value of `cb` from the log at line 26 and that should be equal to the value returned by `malloc` at line 13; a different value for `cb` may lead to a crash in further uses (line 27 and 28).

- Threads created by the implementation of the interposed interface. The operating system, for example, might create threads to deliver events to the application, or might create "anonymous" threads to perform household tasks. The former should be recreated during replay, but the latter not.

- Execution order. Dependencies during recording must be preserved during replay. For example, `ReadFileEx`'s start of an asynchronous I/O must happen before the completion of that I/O event.

2.2 Choosing an Interface

As a starting point for choosing an interface, the developer must choose functions that form a *cut* in the call graph. Consider the call graph in Figure 2. The function `main` calls two functions `f1` and `f2`, and those two both call a third function `f3`, which interacts with the application's environment. The developer cannot choose to have only `f1` (or only `f2`) be the interposed interface. In the case of choosing `f1`, the effects of interactions by `f3` will be recorded only when called by `f1` but not by `f2`. During replay, `f3` interactions caused by `f1` will be read from the log but calls to `f3` from `f2` will be re-executed; `f1` and `f2` will see different interactions during replay.

For faithful replay, the interposed interface must form a cut in the call graph, thus the interface can be `f3` (cut 4) or both `f1` and `f2` (cut 1). Cuts 3 and 4 are also fine, but require R2 to track if `f3` was called from its side of the cut or from the other side. In the case of cut 3, R2 will not record invocations of `f3` by `f2` because `f2`'s invocations will be recorded and replayed. R2 supports all four cuts.

When the interposed interface forms a cut in the call graph, every function is either above the interface or below the interface. For example, if the developer chooses `f1` and `f2` as the interposed interface, then `main` is above and `f3` is below the interface. Functions above the interposed interface will be executed during replay, while functions below the interposed interface will not be executed during replay.

To ensure faithful replay, the cut must additionally satisfy two rules.

RULE 1 (ISOLATION) *All instances of unrecorded reads and writes to a variable should be either be below or above the interposed interface.*

Following the isolation rule will eliminate any shared state between code above and below the interface. A variable below the interface will be unobservable to functions above the interface; it is outside of the debugging scope of a developer. A variable above the interface will be faithfully replayed, executing *all* operations on it. Violating the isolation rule will result in unfaithful replay, because changes to a variable made by functions below the interface will not happen during the replay.

For the Windows API, the isolation rule typically holds. For example, all the operations on a file descriptor are performed through functions rather than direct memory access. During recording R2 records the file descriptor as an integer. During replay, R2 retrieves the integer from the log and returns it to functions above the interposed interface, without invoking the operating system to allocate descriptors. As long as R2 intercepts the complete set of file functions, the recorded file descriptor works correctly with the replayed application and ensures replay faithfulness.

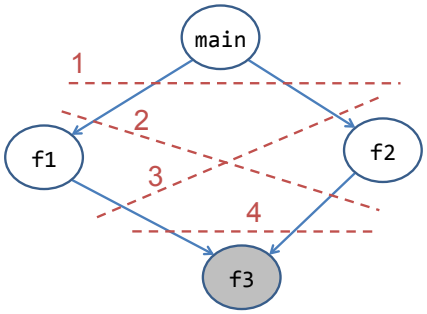

Figure 2: Four cuts in a call graph for record and replay. The function `f3` interacts with the environment.

RULE 2 (NONDETERMINISM) *Any source of nondeterminism should be below the interposed interface.*

If any nondeterminism is below the interposed interface, the impact to functions above the interface will be captured and returned to them. Violating this rule will result in unfaithful replay, because the behavior during replay will be different from during recording.

The sources of nondeterminism are as follows.

1. Calls that receive input data from the external (e.g., environment variables, files, and network).
2. Interprocess communications through shared memory (e.g., `WriteConsole` in Windows communicates with the CSRSS system service for standard input/output through a shared-memory segment).
3. Interactions between threads through shared variables (e.g., spinlocks).

R2 can handle the first source easily if the developer follows the isolation rule because input data from the external is received through functions. For the Windows API the developer must mark these functions being part of the interposed interface, which eliminates the nondeterminism.

For the second source, R2 can re-execute during replay if the replayed application only reads from shared memory. For more general cases the developer must mark the higher-level function that encloses the nondeterminism of shared memory accesses as being part of the interposed interface (e.g., `WriteConsole`).

The third source of nondeterminism stems from variables that are shared between threads via direct memory access instructions rather than functions. A similar

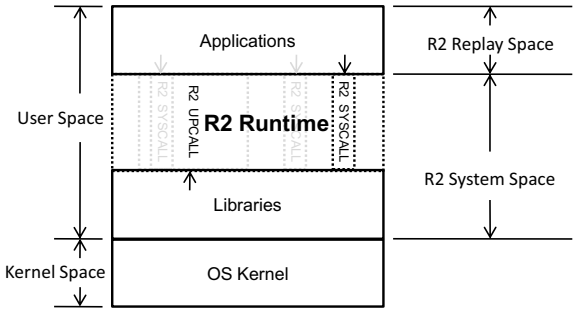

Figure 3: R2 overview. The user space is split into two spaces: R2 runtime that intercepts syscalls and underlying libraries are running in R2 system space; the application executes in R2 replay space.

case is the `rdtsc` instruction on the x86 architecture that reads the CPU timestamp counter. Often these instructions are enclosed by higher-level interface functions (e.g., lock and unlock of spinlocks). Developers must annotate them as being part of the interposed interface.

In practice, library APIs are good candidates for the interposed interface. First, library functions usually have variables shared between internal library routines and it is difficult to select only a subset of them as the interposed interface. Second, most sources of nondeterminism are contained within software libraries (e.g., spinlock variables in a lock library), and they affect external state only via library interfaces.

2.3 Isolation

R2 must address the implementation challenges listed in Section 2.1 for the stubs for the functions that are part of the appropriately-chosen interface. A starting point to handle these challenges is to separate the application that is being recorded and replayed from the code below the interface.

Inspired by isolation between kernel space and user space in operating systems, R2 defines two spaces (see Figure 3): **replay space** and **system space**. Unlike operating systems, however, the developer can decide which interface is the boundary between replay and system space. Like in operating systems, we refer to the functions in the interposed interface as **syscalls** (unless explicitly specified, all syscalls mentioned below are R2 syscalls instead of OS syscalls). Syscalls may register callback functions, which we call **upcalls**, that are issued later into replay space by system space runtime. With these terminologies, we can describe the spaces as follows:

- Replay space. All the code and data that is above the chosen syscall interface.
- System space. The R2 library and the underlying libraries, as well as any application code and data that is below the chosen syscall interface.

R2 records the output of syscalls invoked from application space, the input of upcalls invoked from system space, and their ordering. It faithfully replays them during the replay. R2 does not record and replay events in system space.

Consider the code in Figure 1 again. The developer may have chosen `malloc`, `socket`, `ReadFileEx`, `CreateThread`, `CreateIoCompletionPort`, and `GetQueuedCompletionStatus` as syscalls. The execution of `main` and `WorkerThread` is in replay space and the execution of the syscalls and the underlying libraries is in system space.

To record and replay syscalls and upcalls, R2 generates stubs from their function prototypes. R2 uses Detours [15] to intercept syscalls and upcalls, and detour their execution to the generated stub. For syscalls that take a function as an argument, R2 dynamically generates a stub for the function and passes on the address of the upcall stub to the system layer. This way when later the system layer invokes the upcall, it will invoke the upcall stub.

For syscalls that return data through a pointer argument, R2 must record the data that is returned during recording and copy that data into application space during replay. To do so correctly, the developer must annotate pointer arguments so that R2 knows what data should be recorded and how the stubs must transfer data across the syscall interface. Section 3 describes those annotations.

R2 maintains a replay/system mode bit for each thread to indicate whether the current thread is executing in replay or system space (analog to user/kernel mode bit). When the application in replay space invokes a syscall, the syscall stub sets the replay/system mode bit to system space mode, invokes the syscall, records its results, and restores the mode bit. Similarly, an upcall stub records the arguments, sets the mode bit to replay space mode, invokes the upcall, and restores the mode bit after the upcall returns.

This bit allows R2 to handle a call from system space to a function that is a syscall; if a syscall is called from system space then it must be executed without recording anything (e.g., a call to `socket` from system space). Similarly, if a syscall is called from system space and it has a function argument, then R2 will not generate an upcall stub for that argument. It also allows R2 to apply different policies to different spaces (e.g., allocate memory in a separate pool for code in replay space).

For functions that have state that straddles the boundary between system and replay space (e.g., `errno` in libc), the developer may be able to adjust the interface

to avoid such state (see Section 2.2) or may be able to duplicate the state by linking a static library (e.g., libc) in each space.

2.4 Execution Control

The separation in replay space and system space allows R2 to handle anonymous threads and threads that, for example, the operating system creates to deliver events to the application.

R2 starts as follows. When a user invokes R2 with the application to be recorded, R2's initial thread starts in system space. It loads the application's executable and treats the main entry as an upcall (i.e., the `main` function is turned into an upcall by generating an upcall stub). The stub sets the replay/system mode bit of the current thread to replay space mode, and invokes `main`. R2 assigns the thread a deterministic tag, which the stubs will also record. By this means, R2 puts the functions in the call graph starting from `main` till the syscall interface into replay space.

Anonymous threads that do not interact with the application will be excluded from replay. These threads will not call syscalls and upcalls and thus do not generate log entries during recording and are not replayed. However, if a thread started by the operating system performs an upcall (e.g., to trigger an application registered Ctrl-C handler on Windows), then the upcall stub will set the mode bit; the thread will enter replay space, and its actions will be recorded.

During replay, R2 will replay this upcall, but the thread for the upcall may not exist during replay. R2 solves this problem by creating threads on demand. Before invoking an upcall, R2 will first look up if the current thread is the one that ran the upcall during recording (by comparing the deterministic tag assigned by R2). If not, R2 will create the thread.

For faithful replay, R2 must replay all syscalls and upcalls in the same order as during a record run. In multithreaded programs (and single-threaded programs with asynchronous I/O) there may be dependencies between syscalls and upcalls. Section 4 introduces a few annotations that allow R2 to preserve a correct order.

2.5 Memory Management

If a developer chooses `malloc` and `free` as syscalls, R2 must ensure that addresses returned by `malloc` during recording are also returned during replay. If the addresses returned by `malloc` during replay are different, then during replay the bug that the developer is tracking down might not show up (e.g., invalid value in `cb->buf` will not be reproduced if a different `cb` is returned in Figure 1 because the program crashes before it reaches the buggy state). But, during replay, functions in system space that called `malloc` during recording will not be called during replay, and so `malloc` during replay is likely to return a different value.

To ensure faithful replay application must have an identical trace of `malloc`/`free` invocations to ensure that addresses during recording and replay are the same. R2 uses separate memory allocators for replay and system space. A call to `malloc` allocates memory from a dedicated pool if it is called in replay space (i.e., the mode bit of the current thread is of replay space mode), while it delegates the call to the original libc implementation if it is called in system space.

Memory addresses returned in system space may change due to inherent differences between record and replay, but those addresses are not observable in replay space so they will not impose any problems during replay.

A challenge is addresses allocated in system space but returned to replay space. For example, a syscall to `getcwd(NULL, 0)` to get working directory pathname may call `malloc` internally to allocate memory in system space and return its address to replay space. To ensure replay faithfulness R2 allocates a shadow copy in the dedicated pool for replay space and returns it to the application instead. R2 uses the annotation *xpointer* described in Section 3 to annotate such functions.

Similar to Jockey [28], threads that may execute in replay space have an extra stack allocated from the replay's memory pool, and R2 switches the two stacks on an upcall or syscall. This ensures that the memory addresses of local variables are the same during recording and replay.

Like all other library-based replay tools, R2 does not protect against a stray pointer in the application with which the application accidentally overwrites memory in system space. Such pointers are usually exposed and fixed early in the development cycle.

Resources other than memory (e.g., files, sockets) do not pose the same challenges as memory, as long as the developer has chosen the interposed interface well. R2 does not have to allocate these resources during replay, because the execution in replay space will touch these resources only via syscalls, which R2 records. Memory in contrast is changed by machine instructions, which R2 cannot record.

2.6 Stubs, Slots and Code Templates

R2 generates stubs from code templates. We have developed code templates for recording, replay, etc., but developers can add code templates for other operations that they would like stubs to perform. To allow for this extensibility, a stub is made of a number of slots, with each slot containing a function that performs a specific operation. For example, there is a slot for recording, one for replay, etc.

```
1 int recv (
2     [in] SOCKET s,
3     [out, bsize(return)] char *buffer,
4     [in] int len,
5     [in] int flags );
```

(a) annotated syscall/upcall prototype

```
1 BEGIN_SLOT(record_<?=$f->name?>, <?=$f->name?>)
2     logger << <?=$f->name?>_signature
3            << current_thr_tag;
4     <?if(is_syscall($f)) {?>
5        logger << return_val;<?}?>
6     <?$direction = is_syscall($f) ? 'out':'in';?>
7     <?foreach($f->params as $p) {
8        if ($p->has($direction)) {
9         if ($p->has('bsize')) {?>
10           logger.write(<?=$f->name?>,
11                        <?=$p->val('bsize')?>);
12        <?} else {?>
13    logger << <?=$f->name?>;
14    <?}}}?>
15 END_SLOT
```

(b) record slot function template

```
1 BEGIN_SLOT(record_recv, recv)
2     logger << recv_signature << current_thr_tag;
3     logger << return_val;
4     logger.write(buffer, return_val);
5 END_SLOT
```

(c) generated record slot function

Figure 4: Templates (in PHP [2]) and Slots. R2 uses record (and others like replay) code templates (e.g., (b)) to generate corresponding slot functions (e.g., (c)).

Figure 4 provides an overview of how a record slot function is generated for the `recv` syscall. Developers annotate the prototype of `recv` with keywords from Table 1; for `recv` this step will result in the prototype in Figure 4(a). (On Windows the developer does not have to do any annotation for `recv`, because R2 reuses the SAL annotations.) R2 uses a record template (see Figure 4(b)) to process the annotated prototype and produces the record slot function (see Figure 4(c)).

3 Data Transfers

R2 provides a set of keywords to define the data transfer at syscall and upcalls boundaries. These keywords help R2 isolate the replay and system space. This section presents the data transfer annotations and discusses how R2 uses them to ensure replay faithfulness.

3.1 Annotations

The annotations for data transfers fall into the following three categories.

Direction annotations define the source and destination of a data transfer. In Figure 4, keyword *in* on `s` and `len` indicates that they are read-only and transfer data into function `recv`, while *out* on `buffer` indicates that `recv` fills the memory region at `buffer` and transfers data out of the function. The return value of a function is implicitly annotated as *out*.

```
1  BOOL
2  [prepare(lpOverlapped, lpBuffer)]
3  ReadFileEx (
4      [in] HANDLE hFile,
5      [out] LPVOID lpBuffer,
6      [in] DWORD nNumberOfBytesToRead,
7      [in] LPOVERLAPPED lpOverlapped,
8      [in, callback]
9      LPOVERLAPPED_COMPLETION_ROUTINE completionCb);
10
11 typedef void
12 [commit(lpOverlapped, cbTransferred)]
13 (* FileIOCompletionRoutine) (
14     [in] DWORD dwErrorCode,
15     [in] DWORD cbTransferred,
16     [in] LPOVERLAPPED lpOverlapped );
17
18 BOOL
19 [commit(lpOverlapped, cbTransferred)]
20 GetOverlappedResult(
21     [in] HANDLE hFile,
22     [in] LPOVERLAPPED lpOverlapped,
23     [out] LPDWORD cbTransferred,
24     [in] BOOL bWait );
```

Figure 5: Asynchrony annotations: *prepare* indicates that `ReadFileEx` issues an asynchronous I/O request keyed by `lpOverlapped`, the completion of which is notified as either `FileIoCompletionRoutine` or `GetOverlappedResult`; *commit* indicates the request keyed by `lpOverlapped` is completed and the transferred data size is `cbTransferred`.

Buffer annotations define how R2 should serialize and deserialize data being transferred for record and replay. For `buffer` in a contiguous memory region in Figure 4, which is frequently seen in systems code, keyword *bsize* specifies the size, (e.g., *bsize*(*return*)), so that R2 can then automatically serialize and deserialize the buffer. For other irregular data structures such as linked lists (e.g., `struct hostent`, the return type of `gethostbyname`), R2 requires developers to provide customized serialization and deserialization via operator overloading on streams, which is a common C++ idiom.

Asynchrony annotations define asynchronous data transfers that finish in two calls, rather than in one. For example, in Figure 5 `ReadFileEx` issues an asynchronous I/O request keyed by `lpOverlapped`, which we call a **request key**. Developers use keyword *prepare* to annotate the syscall with the request key and the associated buffer `lpBuffer`. The completion of the request will be notified via either an upcall to `FileIoCompletionRoutine` (line 13) or a syscall to `GetOverlappedResult` (line 20), when the associated buffer has been filled in system space. In either case, developers use keyword *commit* to annotate it with the request key and transferred buffer size `cbTransferred`. R2 can then match `lpBuffer` with its size `cbTransferred` via the request key for record

and replay.

As mentioned in Section 2.5, some syscalls allocate a buffer in system space and the application may use the buffer in replay space. R2 provides the keyword *xpointer* to annotate this buffer, and will allocate a shadow buffer in replay space for the application, at both record and replay time. Data are copied to the shadow buffer from the real buffer in system space during recording, and from logs during replay. While data copy may add some overhead during recording, this kind of syscalls is infrequently used in practice.

Most such syscalls allocate new buffers locally and usually have paired "free" syscalls (e.g., `getcwd` and `free`, `getaddrinfo` and `freeaddrinfo`). Some others without paired "free" functions may return thread-specific or global data, such as `gethostbyname` and `GetCommandLine`. They should be annotated with *xpointer*(*thread*) and *xpointer*(*process*), respectively.

3.2 Code Generation

Figure 4 illustrates the record slot code template and the final record slot function for `recv`. The record slot functions log all the data transmitted from R2 system space to R2 replay space. The slot template (Figure 4(b)) generates code for recording the return value only when processing R2 syscalls (line 4). When scanning the parameters, it will record the data transfer according to the event type and annotated direction keywords (line 6 and 8). Specifically for upcalls, the input parameters and upcall function pointers are recorded so that R2 can execute the same callback with the same parameters (including memory pointers) during replay.

For prototypes annotated with *prepare*, the record slot template will skip recording the buffer. Instead, R2 uses another slot template to generate two extra record and replay slots for each prototype. One is for recording the buffer (including the buffer pointers), and the other is for replaying the buffer, which reads the recorded buffer pointers and fills them with the recorded data. These two slots will be plugged into stubs for prototypes labeled with *commit* at the record and replay phases, respectively. This approach ensures that the memory addresses during replay are identical to the ones returned during recording for asynchronous I/O operations.

4 Execution Orders

For faithful replay, R2 must replay all syscalls and upcalls in the same order as during a record run. For single-threaded programs asynchronous I/O raises some challenges. For multithreaded programs that run on multiprocessors, recording the right order is more challenging, because syscalls and upcalls can happen concurrently, but dependencies between syscalls and upcalls executed by different threads must be maintained. R2 provides developers with two annotations to express such dependencies. This section describes how R2 handles the recording and replay execution order.

```
1  BOOL
2  [sync(hMutex)]
3  ReleaseMutex (
4      [in] HANDLE hMutex );
5
6  DWORD
7  [sync(hMutex)]
8  WaitForSingleObject (
9      [in] HANDLE hMutex,
10     [in] DWORD dwMilliseconds );
```

Figure 6: Syscall-syscall causality annotations using *sync*. R2 serializes the syscall events with the same sync key `hMutex` to obtain an event order, and the causalities between these events are implicitly held by the event sequence.

4.1 Event Definition

In R2 there are three events: syscalls, upcalls, and causalities. R2 uses the causality events to enforce the happens-before relation between events executed by different threads. Consider the following scenario: one thread uses a syscall to put an object in a queue, and later a second thread uses another syscall to retrieve it from the queue. During replay the first syscall must always happen before the second one; otherwise, the second syscall will receive an incorrect result. Using annotations, these causalities are captured in **causality events**. A causality event has a source event e_1 and a destination event e_2, which captures $e_1 \prec e_2$. R2 generates a slot function that it stores in both e_1 and e_2's stubs, which will cause the causality event to be replayed when R2 replays e_1 and e_2.

R2 captures two types of causality events:

- syscall-syscall: a syscall depends on an earlier one, e.g., signal and wait;
- syscall-upcall: a syscall registers a callback that is executed later as an upcall.

To capture syscall-syscall causality, R2 provides the keyword *sync* to annotate syscalls that operate on the same resource. Figure 6 presents an example, where a call to `WaitForSingleObject` that acquires a mutex depends on an earlier call to `ReleaseMutex` that releases it. Developers can then annotate them with *sync*(`hMutex`). We call the mutex `hMutex` a **sync key**. R2 creates causality events for syscalls with the same sync key. In addition to mutexes, a sync key can be any expression that refers to a unique resource. For asynchronous I/O operations (see Figure 5), R2 uses `lpOverlapped` as the sync key.

```
1 HANDLE CreateThread (
2     [in] LPSECURITY_ATTRIBUTES lpThreadAttributes,
3     [in] SIZE_T dwStackSize,
4     [in, callback] LPTHREAD_START_ROUTINE lpStartCb,
5     [in] LPVOID lpParameter,
6     [in] DWORD dwCreationFlags,
7     [out] LPDWORD lpThreadId );
```

Figure 7: A syscall-upcall causality annotation using *callback*. R2 converts the callback argument `lpStartCb` into an upcall stub; when the upcall is delivered, the syscall-upcall causality will be captured.

```
1:  ▷ c(t_0) ← 0
2:  procedure UPDATECLOCKUPONEVENT(e)
3:      if e.type = CAUSALITY_EVENT then
4:          c(t) ← max(c(e.source), c(t)) + 1
5:          c(e) ← c(t)
6:      else
7:          c(t) ← c(t) + 1
8:          c(e) ← c(t)
9:      end if
10: end procedure
```

Figure 8: Algorithm for calculating event clocks. The procedure is invoked when processing every event.

For syscall-upcall causality, developers can use the keyword *callback* to mark the dependency, as illustrated in Figure 7. R2 generates a causality event for the syscall to `CreateThread` and the upcall to `lpStartCb`.

4.2 Recording Event Order

R2 uses a Lamport clock [18] to timestamp all events and uses that timestamp to replay. It assigns each thread t a clock $c(t)$ and each event e a clock $c(e)$. Figure 8 shows R2's algorithm to calculate the Lamport clock for each event.

During recording, R2 first sets the clock of the main thread to 0 (line 1). Then, when an event is invoked, R2 calculates the clock for that event using the procedure UPDATECLOCKUPONEVENT. For non-causality events, R2 simply increases the current thread's clock by one (line 7) and then assigns that value to the event (line 8).

For a causality event, R2 updates the current thread's clock to the greater value of itself and the clock from the source event of the causality (note that $e \leftarrow e.source \prec e.destination$), and increases it by one (line 4). R2 also assigns this value to the causality event (line 5).

When a thread invokes the destination event of a causality event, R2 first runs the slot function for the causality event, which invokes UPDATECLOCKUPONEVENT with the causality event as argument. This will cause the clock of the causality event to be propagated to the thread (line 7 in Figure 8).

There are several possible execution orders that preserve the happens-before relation, as we discuss next.

4.3 Replaying Event Order

R2 can use two different orders to record and replay events: total-order and causal-order. Total-order execution can faithfully replay the application, but may slow down multithreaded programs running on a multiprocessor, and may hide concurrency bugs. Causal-order execution allows true concurrent execution, but may replay incorrectly if the program has race conditions.

4.3.1 Total-Order Execution

Like liblog [10], in total-order execution mode, R2 uses a token to enforce a total order in replay space, including execution slices that potentially could be executed concurrently by different threads. During recording when a thread enters replay space (i.e., returning from a syscall or invoking an upcall), it must acquire the token first and calculate a timestamp if an upcall is present. When a thread leaves replay space (i.e., invoking a syscall), R2 assigns a timestamp to the syscall and then releases the token. On a token ownership switch R2 generates a causality event to record that the token is passed from one thread to another. This design serializes execution in replay space during recording, although threads executing in system space remain concurrent. The result is a total order on all events.

During replay, R2 replays in the recorded total order. As it replays the events, R2 will dynamically create new threads for events executed by different threads during recording (as described earlier in Section 2.4). It will ensure that these threads execute in the same order as enforced by the token during recording. The reason to use multiple threads, even though the execution in replay space has been serialized, is that developers may want to pause a replay and use a standard debugger to inspect the local variables of a particular thread to understand how the program reached the state it is in. In addition, using multiple threads during replay ensures that thread-specific storage works correctly.

4.3.2 Causal-Order Execution

Causal-order execution, however, allows threads to execute in parallel in both replay and system space. R2 does not impose a total order in replay space, it just captures the causalities of syscall-syscall and syscall-upcall. Therefore the application will achieve the same speedup in causal-order execution.

To implement causal-order execution, R2 reuses the replay facility for total-order execution. R2 processes the causal-order event log before replay, uses a log converter to translate the event sequences into any total order that preserves all causalities, and replays using the total-order

execution. If the program has an unrecorded causality (e.g., data race), R2 cannot guarantee to replay these causalities faithfully in causal-order execution. We have not fully implemented the log converter yet, since our focus is replaying distributed applications and total-order execution has been good enough for this purpose.

5 Defining Your Own Syscalls

We have annotated a large set of Win32 API calls for R2 to support most Windows applications without any effort from developers, including those required to be annotated with *xpointer*, *prepare*, *commit*, and *sync*. Sometimes developers may want to define their own syscalls, either to enclose nondeterminism in syscalls (e.g., `rdtsc`, spin lock cases in Section 2.2) or to reduce recording overhead. In this section we use MPI and SQLite as examples to explain how to do this in general.

5.1 MPI

MPI is a communication protocol for programming parallel computing applications. An MPI library usually has nondeterminism that cannot be captured by intercepting Win32 functions (e.g., MPICH [4] uses shared-memory and spinlocks for interprocess communication). To replay MPI applications, this nondeterminism must be encapsulated by R2 syscalls. Therefore, we annotate all MPI functions as syscalls, making the entire MPI library run in system space. Since the MPI library is well encapsulated by these MPI functions, doing this ensures that both rules in Section 2.2 are satisfied.

Annotating MPI functions is an easy task. Most functions only require the *in* and *out* annotations at parameters. Several "non-blocking" MPI functions (e.g., `MPI_Irecv` and `MPI_Isend`) use asynchronous data transfer, which is easily captured using the *prepare* and *commit* annotations. Figure 9 shows the annotated `MPI_Irecv` and `MPI_Wait`, which issue an asynchronous receive and wait for the completion notification, respectively. These functions are associated based on the `request` parameter by the annotations. Section 8.2 presents the number of annotations needed.

5.2 SQLite

SQLite [3] is a widely-used SQL database library. A client accesses the database by invoking the SQLite API. Using Win32 level syscalls, R2 can faithfully replay SQLite client applications. Additionally, developers can add the SQLite API to R2 syscalls so that R2 will record the outputs of SQLite API, and avoid recording file operations issued by the SQLite library in system space.

In certain scenarios, recording at the SQLite API layer can dramatically reduce the log size, compared with recording at the Win32 layer. For example, some

```
1  int
2  [prepare(request, buf)]
3  MPI_Irecv (
4      [out, bsize(MPISize(type, count))] void *buf,
5      [in] int count,
6      [in] MPI_Datatype type,
7      [in] int source,
8      [in] int tag,
9      [in] MPI_Comm comm,
10     [out] MPI_Request *request );
11
12 int
13 [commit(request)]
14 MPI_Wait (
15     [in] MPI_Request *request,
16     [out] MPI_Status *status);
```

Figure 9: Example of asynchrony annotations on MPI functions. The size of the returned buffer at *commit* is by default the registered *bsize* at *prepare* in `MPI_Irecv`.

```
1  int sqlite3_prepare (
2      [in]  sqlite3 * db,
3      [in]  const char * zSql,
4      [in]  nByte,
5      [out] sqlite3_stmt ** ppStmt,
6      [out] const char ** pzTail );
7
8  int sqlite3_column_int (
9      [in] sqlite3_stmt * pStmt,
10     [in] iCol );
```

Figure 10: Example of annotated SQLite functions.

`SELECT` queries may scan a large table but return only a small portion of matched results. For these queries, recording only the final results is more efficient than recording all data fetched from database files. Section 8.4 shows the performance benefits of this approach.

Figure 10 shows two annotated SQLite functions: `sqlite3_prepare` and `sqlite3_column_int` are typical routines for compiling a query and retrieving column results, respectively.

6 Annotations for Optimization

In this section, we introduce two additional annotation keywords to optimize R2's performance.

Cache annotation. By inspecting logs we find that a few syscalls are invoked much more frequently than others—more than two orders of magnitude. Also, most of them return only a status code that does not change frequently (e.g., `GetLastError` on Windows returns zero in most cases). To improve recording performance R2 introduces keyword *cache* to annotate such syscalls. Every time a syscall annotated with *cache* returns a status code, R2 compares the value with the cached one from the same syscall; only when it changes will R2 record the new value in the log and update the cache. An Apache experiment in Section 8.5.1 shows that this optimization reduces the log size by a factor of 17.66%.

Manually Coded Modules	kloc
annotation parser & code generator	4.1
core (interception, isolation, slot)	1.3
upcall (*callback*)	0.7
causality (*sync*)	1.9
aio (*prepare*/*commit*)	1.3
record-replay (memory, data, event)	10.2
Total	19.5

Automatically Generated Modules	kloc
callback.Win32	3.6
causality.Win32	2.9
aio.Win32	1.9
R2.Win32	102.0
R2.app.specific	-
Total	110.2

Table 2: R2 modules.

Reproduce annotation. Some application data can be reproduced at replay time *without* recording. Consider a BitTorrent node that receives data from other peers and writes them to disk. It also reads the downloaded data from disk and sends them to other peers. It is safe to record all input data for replay, i.e., both receiving from network and reading from disk. However, R2 need not record the latter. Developers can use keyword *reproduce* to annotate file I/O syscalls in this case. R2 then generates stubs from a specific code template, to re-execute or simulate I/O operations, instead of recording and feeding. Network I/O, such as intra-group communications of an MPI application, can be reduced similarly [33]. Section 8.5.2 and 8.5.3 show that this optimization can reduce log sizes ranging from 13.7% to 99.4% for BitTorrent and MPI experiments.

7 Implementation

R2 is decomposed into a number of reusable modules. Table 2 lists each module and lines of code (loc). In sum, we have manually written 19 kloc; 110 kloc are generated automatically for R2's Win32 layer implementation.

We have annotated more than one thousand Win32 API calls (see statistics in Table 4). Although this well covers commonly-used ones, Win32 has a much wider interface, and we may still have missed some used by applications. Therefore, we have built an API checker that scans the application's import table and symbol file to detect missing API calls when an application starts.

During replay R2 may still fail because of some unrecorded non-determinisms, e.g., data races not enclosed by R2 syscalls. Since non-determinisms usually lead to different control flow choices thus different syscall invocation sequences, R2 records the syscall signature (e.g., name) and checks it during replay (check whether the current invoked syscall has the same signature with that from the log). By this means, R2 can efficiently detect the mismatch at the first time when R2 gains control after the deviation. When a mismatch is found, R2 reports the current Lamport clock and the mismatch. The developer can then replay the application again with a breakpoint set at the Lamport clock value minus one. When the breakpoint is hit, the developer can then examine how the program reached a different state during replay, and fix the problem (e.g., by adjusting the interposed interface). We have found that this approach works well to debug the R2 interface.

Category	Software Package
web server	Apache, lighttpd, Null HTTPd
database	SQLite, Berkeley DB, MySQL
distributed system	libtorrent, Nyx, PacificA
virtual machine	Lua, Parrot, Python
network client	cURL, PuTTY, Wget
misc.	zip, MPICH

Table 3: Software packages successfully replayed.

8 Evaluation

We have used R2 to successfully replay many real-world applications. Table 3 summarizes an incomplete list. Most of the applications are popular system applications, such as multi-threaded Apache and MySQL servers, which we believe R2 is the first to replay. Nyx [7] is a social network computation engine for MSN and PacificA [19] is a structured storage system similar to Google Bigtable [6], both of which are large, complex distributed systems and have used R2 for replay debugging. The implementation of these applications requires addressing the challenges mentioned in Section 2.

This section answers the following questions.

- How much effort is required to annotate the syscall-upcall interface?
- How important are annotations to successful replay of applications?
- How much does R2 slowdown applications during recording?
- How effective are custom syscall layers and annotations (*cache* and *reproduce*) in reducing log size and optimizing performance?

Similar to previous replay work, we do not evaluate the replay performance, because replay is usually an interactive process. However, the replayed application without any debugging interaction runs much faster

Interface	#func	*in*	*out*	*bsize*	*cb*	*xptr*	*pr*	*ci*	*sync*	*cache*	*reproduce*	#serial	kloc
Win32	1,301	1,100	631	168	53	11	17	4	30	2	3	7	110.2
MPI	191	171	150	20	6	4	6	4	1	0	4	5	22.2
SQLite	153	150	16	4	19	3	0	0	7	0	0	0	15.7

Table 4: Information about annotations and code generation. Columns with annotation keywords show the number of functions for each keyword. Keywords *callback*, *xpointer*, *prepare*, *commit*, are abbreviated to *cb*, *xpr*, *pr*, *ci*, respectively. The last two columns list the number of functions that require customized (de)-serialization, and lines of automatically generated code, respectively.

than when recording (e.g., a replay run of BitTorrent file downloading is 13x faster).

8.1 Experimental Setup

All experiments are conducted on machines with 2.0 GHz Xeon dual-core CPU, 4 GB memory, two 250 GB, 7200 /s disks, running Windows Server 2003 Service Pack 2, and interconnected via a 1 Gbps switch. Unless explicitly specified, the application data and R2 log files are kept on the same disk; the record run uses total-order execution; all optimizations (i.e., cache and reproduce) are turned off.

8.2 Annotation Effort

We annotated the first set (500+ functions) of the Win32 syscall interface within one person-week, and then annotated as needed. We reused *in*, *out*, *bsize*, and *callback* from the Windows Platform SDK, and manually added the other annotations (i.e., *xpointer*, *prepare*, *commit*, *sync*); we manually annotated only 62 functions.

We found that once we decided how to annotate a few functions for a particular programming concept (e.g., asynchronous I/O, or synchronization), then we could annotate the remaining functions quickly. For example, after we annotated the file-related asynchronous I/O functions, we quickly went through all the socket related asynchronous I/O functions.

For the two other syscall interfaces, MPI and SQLite (discussed in Section 5), we spent two person-days annotating each before R2 could replay MPI and SQLite applications. The first four keywords (*in*, *out*, *bsize*, and *callback*) are trivial and cost us little time, and we mainly spent our time on other annotations and writing customized (de)-serialization functions.

Table 4 lists for the three syscall interface the annotations used, how many functions used them, the number of functions that needed customized (de)-serialization, and the lines of code auto generated (approximately 148 kloc). The table shows that the annotations are important for R2; without them it would have been a tedious and error-prone job to manually write so many stubs.

Configuration	Request#/s	Slowdown	Log rate
native	1242.23	-	-
stub only	1241.75	0.04%	-
log	1125.58	1.34%	0.760
causal-order	1197.52	3.60%	1.114
total-order	1129.94	9.04%	0.781

Table 5: Apache performance under different R2 configurations (cache on). Log rate is measured in KB/req. Client concurrency level is 50 and the download file size is 64 KB.

8.3 Performance

We measure the recording performance of R2 using the Apache web server 2.2.4 with its default configuration (250 threads) and the standard ApacheBench client, which is included in the same package.

Table 5 shows the reduction in request throughput and the log overhead under different R2 configurations. We use ApacheBench to mimic 50 concurrent clients, all of which download 64 KB static files (which is a typical web page size). Each configuration in the table executes 500,000 requests. As we can see, the stub, the logger, and the causal-order execution have little performance impact; the total-order execution imposes a slowdown up to 9.04%, which we believe to be acceptable for the purpose of debugging. The log produced for each request is approximately 0.8 KB, slightly bigger for causal-order execution mode since it needs to log more causality events.

Figure 11 shows the results for total-order and causal-order configurations, with a varied number of concurrent clients and file size. We see that when the download file is larger, the slowdown is smaller. This is because the larger file size means that the execution in replay space costs less CPU time, and the slow down imposed by total order execution is less. For the smallest size of 16KB we tested, the average slowdown for all concurrency levels is 11.2% under total-order configuration and 4.9% under causal-order configuration.

In addition to Apache, we have also measured the performance of many other applications while recording. The slowdown for most cases is moderate (e.g., 9% on

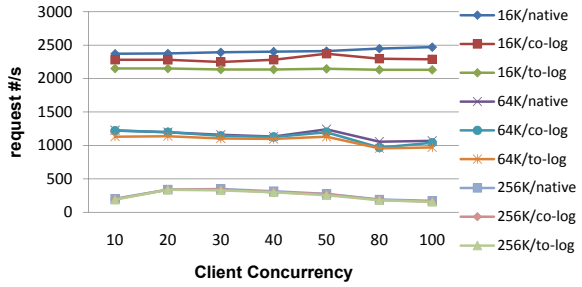

Figure 11: Apache performance using total-order and causal-order configurations, with a varied number of concurrent clients and file size.

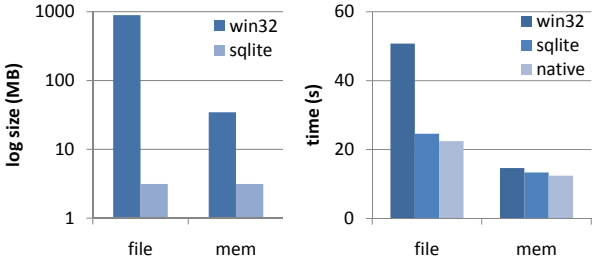

Figure 12: SQLite log size and execution time at Win32 and SQLite interfaces using FILE and MEM configurations.

average for the standard MySQL benchmark [12]). There are exceptions, such as the SQLite case below. The performance of these exceptions, however, can be improved using either customized syscall layer or optimization annotations.

8.4 Customized Syscall Layers

This section evaluates the performance of R2 for SQLite using two syscall layers, i.e., Win32 and SQLite, which was discussed in Section 5.2.

We adopted a benchmark from Nyx [7], which calculates the degree distribution of user connections in a social network graph. The calculation is expressed as a query: `SELECT COUNT(*) FROM edge GROUP BY src_uid`. We perform the query 10 times, and measure log size as well as execution time. The data set contains 156,068 edges and is stored in SQLite; the file size is approximately 3 MB.

By default SQLite stores temporary tables and indices in files; it can store them in memory by setting an option. We use FILE and MEM to name the two configurations. Other options are default values.

Figure 12 shows log size and execution time for recording at the two syscall layers, respectively. In either configuration, recording at the Win32 interface produces much larger logs (890 MB and 35 MB), compared to recording at the SQLite interface (only 3 MB). The slowdown factors of recording at the two interfaces are 126.3% and 9.6% under FILE, 17.8% and 7.3% under MEM, respectively.

Note that the recording at the SQLite interface produces the same size of log for the two configurations, because the SQL layer does not involve file I/O and the log size is not effected by the configurations.

From these results we can see that recording at the SQLite layer can reduce log overhead and improve performance, if for a query SQLite must perform I/O frequently.

Cached Syscalls	Call#	Miss#	Hit Ratio
`GetLastError`	618,015	99,948	83.82%
`CloseHandle`	150,016	2	99.99%
`setsockopt`	150,003	1	99.99%
`FindClose`	100,147	2	99.99%
`WaitForSingleObject`	100,014	4	99.99%
Total	1,118,195	99,957	91.06%

Table 6: Apache cached syscalls with cache miss and hit statistics. Client concurrency level is 50 and download file size is 64 KB.

8.5 Optimization Annotations

As discussed in Section 6, R2 introduces two annotation keywords to improve its performance. We evaluate them in this section.

8.5.1 The cache Annotation

We use the Apache benchmark again to evaluate the cache annotation. The experiment runs R2 in total-order execution mode. The client's concurrency level is 50 and the file downloaded is 64 KB. Profiling Apache shows that 5 out of 61 syscalls contribute more than 50% of syscall. We use the cache annotation for these syscalls. Table 6 shows how many times these syscalls were invoked and did not hit the cache in one test run. We see that the return values of these syscalls were mostly in the cache, and that the average hit ratio is 91.06%. This reduced the log size from 21.99 MB to 18.1 MB (approximately 17.66% reduction). We applied the cache optimization to only five syscalls, but we could gain more benefits if we annotated more syscalls.

8.5.2 Reproduced File I/O

As discussed in Section 6, when the reproduce annotation is used for file I/O when recording BitTorrent, the file content that is read from a disk is not recorded, and the related file syscalls are re-executed during replay.

We use a popular C++ BitTorrent implementation libtorrent [1] to measure the impact of this annotation. The experiment was conducted on 11 machines, with one

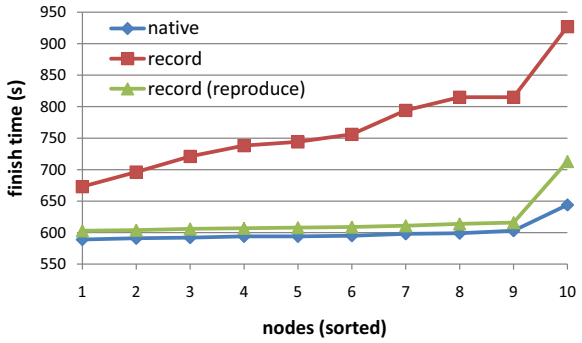

Figure 13: Finish time of 10 BitTorrent nodes in runs of native, record without and with reproduced I/O optimization.

seed and 10 downloaders. The seed file size was 4 GB; the upload bandwidth was limited to 8 MB/s. R2 ran in total-order mode with cache optimization off.

The average log sizes of the recording run without and with the reproduce annotation are 17.1 GB and 5.4 GB, respectively. The optimization reduces the log size by 68.2%. Relative to the 4 GB file size, the two cases introduce 297.5% and 26.4% log size overhead, respectively.

Figure 13 presents the finish time of the 10 downloaders for a native run, as well for recorded runs without and with the reproduce annotation. On average, the slowdown factors of recording without and with the annotation are 28% and 3%, respectively. We can see that the *reproduce* annotation is effective when recording I/O intensive applications, reducing both the log and performance overhead.

8.5.3 Reproduced Network I/O

We use the MPI syscall layer to evaluate the benefit of the *reproduce* annotation for network I/O. The experiment was conducted in our MPI-replay project [33], which uses R2. We annotated MPI functions using the *reproduce* annotation so that the messages are not recorded but reproduced during replay. Table 7 shows the effectiveness for two typical MPI benchmarks: GE [14] and PU [11]. We see that the client process of PU gains much benefit from this keyword; the log size is reduced by more than 99.4%. For GE, it also results in a log size reduction, but of about 13.7%.

	w/o opt (MB)		w/ opt (MB)		Ratio
	node 0	node 1	node 0	node 1	
GE	55.3	55.3	47.7	47.7	86.3%
PU	4.5	1170.0	5.7	1.7	0.6%

Table 7: Reproduced network I/O optimization on MPI. "Ratio" is the log size with this optimization compared that without. The other fields are R2 log size on each node.

9 Related Work

R2 borrows many techniques from previous replay tools, in particular from library-based ones. This section relates R2 to them in more detail.

Library-based replay. Several replay tools use a library-based approach. The closest work is Jockey [28] and liblog [10], where a runtime user-mode library is injected into a target application for record and replay. We borrow many ideas from these tools (e.g., using a token to ensure total-order execution) but extend the library-based approach to a wider range of applications using stricter isolation (inspired by operating system kernel ideas), by flexible customization of the record and replay interface, by annotations, and automatic generation of stubs.

R2 also isolates the application from the tool in a different way. For example, Jockey tries to guarantee that the application behaves the same with and without record and replay, and liblog shares the same goal. Consequently, they both send the memory requests from the *tool* to a dedicated memory region to avoid changing the memory footprint of the application. As discussed in Section 1, R2 aims for replay faithfulness instead, and it manages memory requests from the *application*.

Jockey and liblog have a fixed interface for record and replay (a mix of system and libc calls); any nondeterminism that is not covered will cause replay to fail. R2 enables developers to annotate such cases using keywords on functions of higher-level interfaces to enclose nondeterminism.

On the implementation side, both Jockey and liblog have manually implemented many stubs (100+); R2's more automatic approach makes it easier to support a wider range of syscalls. For example, Jockey does not support multithreading; liblog also does not support asynchronous I/O and other functions.

RecPlay [25] captures causalities among threads by tracking synchronization primitives. R2 uses that idea too, but also captures other causalities (e.g., syscall-upcall causalities). RecPlay uses a vector clock during replay and can detect data races. This feature could be useful to R2 too.

Another library-based approach but less related is Flashback [29], which modifies the kernel and records the input of the application at system call level. Since it is implemented as a kernel driver, it is less easy to deploy and use than R2.

Domain-specific replay. There are a large number of replay tools focusing on applications using restricted programming models, such as distributed shared mem-

ory [27] or MPI [26], or in specific programming languages such as Standard ML [30] or Java [17]. This approach is not suitable for the system applications that R2 targets. In fact, we built a replay tool before [21], which relies on the programmers to develop their applications using our own home-grown API. The limitation of this work propelled us to design and build R2.

Whole system replay. A direct way to support legacy applications is to replay the *whole* system, including the operating system and the target applications. A set of replay tools aim at this target, either using hardware support [32, 24, 23] or virtual machines [8, 16, 5]. They can replay almost every aspect of an application's environment faithfully, including scheduling decisions inside the operating system, which makes them suitable to debug problems such as race conditions. They can achieve similar performance as R2; ReVirt [8], for instance, has a slowdown of 8% for rebuilding the kernel or running SpecWeb99 benchmarks. However, they can be inconvenient and expensive to deploy. For example, developers must create a virtual machine and install a copy of the operating system to record and replay an application.

Annotations. Annotations on functions are widely used in many fields. For example, a project inside Microsoft uses SAL and static analysis to find buffer overflows [13]. Instead, SafeDrive [34] inserts runtime checks where static analysis is insufficient according to the annotations. While they all focus on finding bugs, R2 uses annotations to understand function side effects, and generates code to record and replay them.

Enforcing isolation with binary instrumentation. XFI [9] is a protection system which uses a combination of static analysis with inline software *guards* that perform checks at runtime. It ensures memory isolation by introducing external checking modules to check suspicious memory accesses at runtime. Because XFI monitors the memory access at instruction level, its overhead varies from 5% to a factor of two, depending on how the static analysis works and also the benchmark. R2 isolates at function interfaces and targets replay, which allows it to be more loose in its isolation in some ways (i.e., it does not have to protect against attacks), but more strict in other ways (i.e., memory addresses cannot change from recording to replay).

10 Conclusion

R2 uses kernel ideas to split an application's address space into a replay and a system space, allowing strict separation between the application and the replay tool. With help from the developer, who specifies some annotations on the syscall interface, R2 carefully manages transitions between replay and system space at the syscall interface, and isolates resources (e.g., threads and memory) within a space.

The annotations also allow R2 to generate syscall and upcall stubs from code templates automatically, and make it easy for developers to choose different syscall/upcall interfaces (e.g., MPI or SQLite). It also allows developers to enclose nondeterminism and avoid shared state between replay and system space. Annotations for optimizations can reduce the record log size and improve performance.

By using these ideas R2 extends recording and replay to applications that state-of-the-art library-based replay tools cannot handle. R2 has become an important tool for debugging applications, especially distributed ones, and a building block for other debugging tools, such as runtime hang cure [31], distributed predicate checking [20], task hierarchy inference [22], and model checking.

Acknowledgment

We thank Alvin Cheung, Evan Jones, John McCullough, Robert Morris, Stefan Savage, Alex Snoeren, Geoffrey Voelker, our shepherd, David Lie, and the anonymous reviewers for their insightful comments. Thanks to our colleagues Matthew Callcut, Tracy Chen, Ruini Xue, and Lidong Zhou for valuable feedback.

References

[1] libtorrent 0.11. `http://libtorrent.sourceforge.net/`.

[2] PHP: Hypertext preprocessor. `http://www.php.net/`.

[3] SQLite 3.5.8. `http://www.sqlite.org/`.

[4] D. Ashton and J. Krishna. *MPICH2 Windows Development Guide*. Argonne National Laboratory, 2008.

[5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] Y. Chen, T. Chen, M. Chen, and Z. Zhang. Islands in the MSN Messenger buddy network. In *SocialNets*, 2008.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[9] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[10] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX*, 2006.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[12] Z. Guo, X. Wang, X. Liu, W. Lin, and Z. Zhang. Towards pragmatic library-based replay. Technical Report MSR-TR-2008-02, Microsoft Research, 2008.

[13] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[14] Z. Huang, M. K. Purvis, and P. Werstein. Performance evaluation of view-oriented parallel programming. In *ICPP*, 2005.

[15] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium*, 1999.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[17] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *IPDPS*, 2000.

[18] L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[19] W. Lin, M. Yang, L. Zhang, and L. Zhou. PacificA: Replication in log-based distributed storage systems. Technical Report MSR-TR-2008-25, Microsoft Research, 2008.

[20] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging deployed distributed systems. In *NSDI*, 2008.

[21] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *NSDI*, 2007.

[22] H. Mai, C. Gao, X. Liu, X. Wang, and G. M. Voelker. Towards automatic inference of task hierarchies in complex systems. In *HotDep*, 2008.

[23] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using Strata. In *ASPLOS*, 2006.

[24] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[25] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *TOCS*, 17(2):133–152, 1999.

[26] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay for an MPI-based multi-threaded runtime system. In *ParCo*, 1999.

[27] M. Ronsse and W. Zwaenepoel. Execution replay for treadmarks. In *PDP*, 1997.

[28] Y. Saito. Jockey: A userspace library for record-replay debugging. In *AADEBUG*, 2005.

[29] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.

[30] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[31] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang. Hang analysis: Fighting responsiveness bugs. In *EuroSys*, 2008.

[32] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[33] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. MPIWiz: Subgroup reproducible replay of MPI applications. In *PPoPP*, 2009.

[34] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.