Towards Pragmatic Library-based Replay

Zhenyu GUO
Microsoft Research Asia
zhenyu.guo@microsoft.com

Xi WANG
Tsinghua University
wangxi01@mails.tsinghua.edu.cn

Xuezheng LIU
Microsoft Research Asia
xueliu@microsoft.com

Wei LIN
Microsoft Research Asia
weilin@microsoft.com

Zheng ZHANG
Microsoft Research Asia
zzhang@microsoft.com

Abstract

Deterministic replay is a powerful approach for debugging multi-threaded and distributed applications that involve complex non-determinism. Recently, several replay tools based on user-space libraries were proposed. Such library-based tools are lightweight and efficient, and focus on the application being debugged. However, several significant problems remain, such as in-address-space tool interference, absence of support for advanced features in modern operating systems (e.g. asynchronous I/O), and error-prone engineering efforts for wrapping a large set of API functions to enable logging and replaying.

We propose three mechanisms to cope with these problems. First, we resolve the tool interference problem by space separation, using a virtual execution layer composed of intercepted functions. The virtual execution layer isolates both memory and execution of the application from supporting libraries and in-address-space replay tool, hence systematically removes the interference. Second, based on an event-view of the execution, we are able to handle advanced operating features such as asynchronous I/O and exceptional control flow (e.g. signal). Third, instead of manually coding the wrappers, we use an annotation-aware code generator so that the process of producing the wrappers is more robust and efficient. We have handled more than 750 API functions and the code generator is responsible for more than 37,000 lines of wrapper code.

To the best of our knowledge, these three mechanisms, space separation, event-based replay, and annotation-aware code generation are novel for library-based deterministic replay. We have successfully replayed many large and complex software packages with low overhead, including both server and distributed system, including those developed by ourselves and by the community. MySQL and Apache, for instance, experienced around 10% slowdown.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.5 [*Operating Systems*]: Reliability; D.4.9 [*Operating Systems*]: Systems Programs and Utilities

General Terms Experimentation, Performance, Reliability

Keywords Replay, Space Separation, Event, Annotation

1. Introduction

Modern systems are increasingly complex to debug. They are usually multi-threaded, employ advanced features such as asynchronous I/O to gain a performance edge, and are often distributed. Their execution embodies a great number of non-determinisms, and thus posts great challenges to the traditional cyclic debugging process, in which developers repeatedly run the same program with the same input and gradually locate the root causes of bugs. This is because features such as multi-threading and asynchronous calls can result in different behaviors across debugging sessions. A distributed, multi-threaded application amplifies the problem even further. Developers must instantiate all instances and the states are distributed as well.

Deterministic replay is not a new technology, but becomes extremely relevant in the above context. It removes non-determinism in the debugging session, and thus re-enables cyclic debugging. It can bring together all relevant states originally spread across machines in a distributed system and, if the developer chooses to, debug replay any individual (or a subset of) instance(s). As the programming model evolves, deterministic replay also requires new development.

Our target scenario is to debug the application *only*. As such, previous approaches that replay the whole system at the level of virtual machine are unnecessarily heavy. For a more complete treatment of existing work, we refer the readers to Section 2. Recently, the library-based approach has been proposed as an interesting alternative, which we adopt in this work as well. Tools such as Jockey [19] and liblog [8] inject a runtime library into the target application and intercept libc API functions. These functions are then redirected to function-specific wrappers, that log the output at logging time and feed them back at replay. The library-based approach is considered efficient and lightweight enough to be always-on even at deployed distributed systems [8].

This paper describes our library-based replay tool constructed out of three novel mechanisms, resolving issues that were not previously well addressed.

First, the library-based tool executes in the same address space of the application, and thus interferes with the target application. In other words, the execution of the tool itself brings in non-determinism across logging and replay. We solve this problem by forming the intercepted functions into a virtual execution layer, confining the application logic in the *application subspace*, moving the tool execution as well as the services provided by supporting libraries and operating system in the *system subspace*. This virtual

execution layer isolates both the memory management as well as the execution of the application, and systematically removes the interference. Previous approaches solved part of the problem of tool interferences in an ad hoc way.

Second, some advanced features in modern operating system, such as asynchronous I/O, are not addressed in previous work. These features are critical to improving the performance of application. However, their asynchronous nature is harder to reason and can be responsible for more bugs. We suggest that the execution be broken down into a few classes of discrete events. We assign clock values to the events that fully respect the happens-before relationship. This mechanism simplifies the complex control flow in a system, and enables event-based replay. Under this model, asynchronous I/Os are ordinary events that occur at their recorded clocks.

Enabling the above two features requires the wrapping of many API functions. Instead of manually coding these wrappers, we automatically generate wrapper code based on annotations. While some annotations use new types specifically tailored to this tool, the majority of them are already available from their prototypes in the header files. This process is much more robust and efficient. We can quickly adjust the layer of interception by annotating a higher-level API call that calls a number of lower-level API functions. We have handled more than 750 API functions and the code generator is responsible for more than 37,000 lines of wrapper code.

To the best of our knowledge, these three mechanisms, space separation, event-based replay, and annotation-aware code generation are novel for deterministic replay. We also believe that none of them is dispensable to building a pragmatic library-based replay tool. We have successfully replayed many large and complex software packages with low overhead, including both server and distributed system, those developed by ourselves and by the community. MySQL and Apache, for instance, have around 10% slowdown using their own benchmarks.

The rest of the paper is organized as follows. We go through related work in Section 2, paying special attention to the challenges of library-based replay tools. We then give the overview design of our replay tool in Section 3. Next, the design and implementation of the three key solutions mentioned above are thoroughly discussed in Section 4, 5, and 6. Section 7 presents results of our evaluation and Section 8.1 offers a summary. We conclude in Section 9.

2. Challenges and Related Work

There is a large body of existing work in the area of deterministic replay. A number of tools focus on applications using restricted programming model such as distributed share memory [18] or MPI [17], or in specific programming languages such as Standard ML [21] or Java [12]. Our previous work that performs predicate checking for distributed applications to identify subtle bugs [13] also relies on a replay tool; the system that are debugged must be developed using our own home-grown API functions. Nevertheless, we were able to identify many challenging bugs, including one in well-known, previously proven specification. In fact, it is the lessons that we gained from that work which propel us to propagate the value of the replay to applications developed using native API functions.

To enable wider applicability, many attempts have been made to replay legacy applications, which were written mostly in C and C++. In general, they fall into two categories. The first set of replay tools replay the *entire* system, either using hardware support [22, 15, 14] or virtual machine [7, 11]. As a result, they pay the overhead of including components unrelated to the application that is under debugging.

In contrast, our replay tool targets the scenario where the underlying infrastructure (i.e. system libraries and operating system) is regarded as trusted, and the application running on top of the infrastructure is what developers care about. One example is FlashBack [20], which logs applications at system call level, and hence needs host operating system modification.

We took the same approach as Jockey [19] and liblog [8], in which a runtime library is injected into target application to intercept library functions. Compared to other approaches, this method is efficient and lightweight, and has the potential to be always-on when the application is deployed. Our contributions lie in a set of technologies that resolve a number of challenging issues not well addressed before. In the following, we will discuss these issues in detail.

2.1 Tool Interference

One of the most obvious drawbacks of a library-based tool is that the tool lives and executes in the same address space as the target application. This results in memory-related non-determinism in a number of ways. First, the replay tool itself needs memory, and the memory request sequence is entirely different at logging time and replay time. Second, the intercepted functions are skipped during replay and if they had allocated memory during the logging phase, such memory requests simply disappear. Third, a buggy program can crash the replay tool by overwriting data structures internal to the tool. Previous approaches solve some of these issues. For instance, Jockey [19] allocates a dedicated memory pool for the replay tool and switches stacks every time when a thread enters or leaves an intercepted function. Authors of liblog [8] use Purify [4] to check target application for corruption. We adopt a more general approach. Analogous to how modern operating system establishes trust boundary, we use thread-specific flag to distinguish two subspaces. The application space hosts the target application and requests memory from a dedicated memory manager that has no random effect. This occurs while the wrappers of the intercepted API functions, as well as the replay tool, run in system space. We also adopt the same approach as Jockey to deal with potential data corruption by switching stacks from system and application space for a thread.

Both the replay tool and the intercepted functions may call functions that are already intercepted (e.g. write or read a file, allocate memory, etc.). To avoid infinite recursion, previous works pays a lot to implement their own set of API functions. Meanwhile, our space separation mechanism directs the call to the native implementation when the invocation is inside system space. We believe this solution is far more natural and elegant than previously proposed ones. Section 4 provides the full details.

2.2 Inherent Non-determinism

While tool interference is a challenge, it is specific to the library-based approach. There are inherent non-determinisms that are independent of how we replay.

First, application can have different control flows between the logged run and the replayed run. This generally includes issues about multi-threading and multi-machine, as well as some other advanced features on modern operating systems, such as asynchronous I/O and various exceptional control flows (signal, etc.). In fact, no previous work has handled asynchronous I/O, a feature that critically improves application performance. Jockey works only for single-thread application and does not replay distributed system. We note that liblog employs cooperative scheduling to handle multi-threading. In contrast, we segment thread execution into several types of events, including ones that handle asynchronous I/O and exceptional control flow. We assign logical time stamp to events so that when they are processed in order, the full internal state of the application can be reconstructed on any of the processes. Different algorithms are developed to deal with multi-

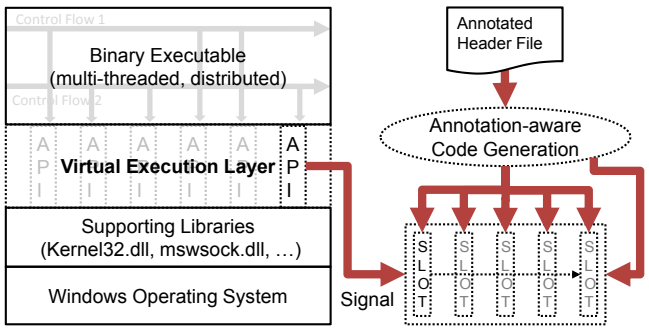

Figure 1. Overview of our replay tool

threading and data race, as we can see in Section 5. While still having room to improve, we believe such *event-based* replay methodology has significantly improved the state-of-the-art.

Another interesting yet challenging problem is anonymous threads in target processes. We have observed that there are always two other threads we did not know about in a process on a Windows XP SP2 machine. These threads belong to neither the target application nor the replay tool. We speculate that they are from some system software such as the firewall. Therefore, we put them into system space, and thus avoid having to log and replay them.

Second, environmental factors such as time, file, and network, can have different behaviors at replay time. The solution, as many of the previous work adopts, is quite straightforward. We log the output at log time and feed it back at replay. However, this can also be challenging under some circumstances. For instance, the x86 instruction `rdtsc` reads time stamp counter and `rdpmc` reads performance monitoring counter. Jockey takes the heavyweight approach to scan the assembly and wraps these instructions into functions. In addition, modern operating systems use Local Procedure Call (LPC) for the application to communicate with some system services. These services are typically very complex and it is extremely challenging to deal with them in a clean way. Our approach is to instead intercept the API functions that are composed of these low-level instructions/functions, and therefore hide their complexities. The consequence is that once we move up the interception level, the API surface gets much wider. Thus, we develop an *annotation-based automatic code generator* to produce wrappers based on annotated function prototypes. We have intercepted 751 functions, compared with 78 and 200+ from Jockey and liblog, with a total of more than 37K lines of generated code.

The third problem is memory footprint inconsistency. We have already discussed how we deal with the inconsistencies caused by tool interference in Section 2.1. Furthermore, uninitialized memory is another source of non-determinism. Previous work fills memory blocks with zero [8, 19] as a solution. Instead, we fill these blocks with 0xcd to avoid hiding some potential bugs.

3. Overview

Figure 1 depicts the overview of our replay tool. We can see that upper applications communicate with underlying libraries and operating system via lots of API functions. We form a *virtual execution layer* by intercepting these functions with Detours [10]. This is reasonable because API functions represents a natural boundary of a program and the underlying supporting infrastructure, including various libraries and operating system. We target this tool for debugging application logic, which assumes that the underlying libraries and operating system are correct, efficient, and robust.

This virtual execution layer establishes two subspaces. Above it is the *application space*, and below it is the *system space*. All logic of our replay tool (including that of both logging and replaying) resides in system space, along with all supporting libraries and the operating system. As we will describe in much more detail in Section 4.1, the concept of space separation is key to eliminate tool interference as well as many other artifacts such as memory footprint inconsistency.

Each intercepted API function is converted into a flexible extension called SignalEx object. A SignalEx object treats a function invocation as a signal-slot process, a concept we borrow from Qt [6]. In such a process, a signal is an event publisher, which is connected to several slots kept in a linked list; each slot is an event subscriber. As the signal triggers, the slots are executed one by one. The linked list can be dynamically reconfigured. For instance, logging and replay are distinctive slots that we plug into the SignalEx of any intercepted API; the logging slot records the output of an API invocation, whereas the replay slot feeds from the log in place of the native implementation of the API.

This model also makes our replay tool *event-based*, in which the complete execution of an application is broken down into several types of events. Our replay tool dynamically assigns clock values to these events such that they are processed in order at replay time. Besides, advanced OS features such as asynchronous I/O as well as exceptional control flow are also well handled in our model. Section 5 describes how we implement the event-based replay in detail.

Instead of hand-coding the API function wrappers, we use *annotation-aware code generation* to produce code snippets that convert API functions into SingalEx object, and also to generate slots that are target to various functions (e.g. logging or replay). This mechanism is far more robust and reliable, as our experience has proved.

We share the view that logging should have low overhead [19, 8] so that it is possible to be always-on in a deployed system. To avoid contention among threads, log files are per thread. We also use a dual-buffer to overlap I/O to the log files with the logging. We will demonstrate in Section 7 that the logger is quite lightweight. We are implementing checkpoint to truncate logs, which is important for long-running and I/O intensive systems. Note that we only need to make checkpoint for application space, which is much simpler to do compared to that for the whole process. This is another advantage that space separation brings.

Our tool relies on three key mechanisms: space separation, event-based logging and replay, and annotation-aware code generation. Our experience is that none of the three is dispensable, and this is demonstrated by the successful and low-overhead log and replay of many complex and large systems, which has not been achieved before. Table 1 presents challenges and approaches from previous work and our replay tool; reference to related sections for the approaches is given.

4. Space Separation

The idea of space separation is to use the intercepted API functions as a layer that "tighten" the surface of what ought to be logged and replayed. As we discussed earlier, the application being replayed lives in application space above the virtual execution layer, and all logic of our replay tool (including that of both logging and replaying), along with all supporting libraries and the operating system resides in system space, which is below the virtual execution layer. In what follows, we will first describe how we implement space separation. Next, we will describe the main strength of space separation, which enables us to apply different polices to each of the space. The approach allows us to isolate memory consumption of user code and completely avoiding problems such as inconsistent memory footprints in application space.

Challenge (mostly non-determinisms)	Approaches proposed by previous work	Our approach
memory ops from replay tool	dedicated memory pool	space separation (memory isolation in Sec 4.2)
memory ops inside wrapped functions	unknown	space separation (memory isolation in Sec 4.2)
program bugs like buffer overflow	stack switch; purify	space separation (stack switch in Sec 4.2)
recursive call	dedicated code base for the tool	space separation (Sec 4)
multi-threading	limit parallelism	event-based replay (Sec 5)
data race (including share memory)	limit parallelism; detect at replay time	event-based replay (Sec 5)
asynchronous I/O	unknown	event-based replay (Sec 5)
exceptional control flow	delay signal handler at syscall boundaries	event-based replay (Sec 5)
anonymous threads	unknown	space separation (Sec 4)
low level complexity(e.g. rdtsc/rdpmc)	scan/wrap these instructions	annotation-aware code generation (Sec 6)
low level complexity(e.g. LPC)	unknown	annotation-aware code generation (Sec 6)
memory consistency for application	unknown	space separation (memory isolation in Sec 4.2)
memory randomization	close randomization	space separation (memory isolation in Sec 4.2)

Table 1. Challenges and approaches from Past and Present. The challenges are classified into four groups. The first group is related to the library-based approach; the remaining three are related to control flow, environment, and memory, respectively.

4.1 Implementation

We first classify the execution flows inside the target application process with our library-based replay tool into four categories, so as to make it clear what should be in each space. They are listed below.

- Execution flow from application threads, excluding that of the wrapped API functions (the wrapper).
- Execution flow from the wrappers, which serves both the application (e.g. invocation of the native implementation of a call during logging) and the replay tool (e.g. executing the extension slots).
- Execution flow from threads purely in the replay tool. A good example is the dedicated thread that asynchronously flush log buffer to disk.
- Execution flow from anonymous threads that are neither related to the application nor the replay tool, as mentioned in Section 2.

In principle, what application space should contain is only the first category, nothing more and nothing less.

Since a thread can be in either system space or application space, we use a thread specific flag to identify which space the current thread belongs to. When our replay tool first takes over the application, we create dedicated threads for our replay tool. Then, we default all threads in system space, and explicitly set a flag for the main application thread so that it is in application space. After that, all threads created in application space will be in application space, and vice versa. This is done by wrapping the start routine of new threads and setting thread specific flags outside of the start routine. Up to now, we have already separated the execution flows from various threads into two spaces. However, execution flows of wrappers in application thread are still in application space. Therefore, we embed a space switch action in the wrappers, as is shown in the pseudo code of the wrapper below.

```
1 int __wrapper_foo(int param1, int param2)
2 {
3   if (IsThreadInSystemSpace())
4     return CallNativeFoo(param1, param2);
5   else {
6     SetThreadInSystemSpace();
7     int retVal = SignalExOfFoo.Execute(param1, param2);
8     SetThreadInApplicationSpace();
9     return retVal;
10 }
11}
```

As we can see, the execution is separated into system space and application space with a thread local flag (line 3). Invocations of the function from system space are dispatched to the native implementation of the function (line 4). However, when it is invoked from application space, the thread is switched to system space (line 6), and the execution is dispatched to SignalEx object we mention to process the extensions (line 7). When the execution of a SignalEx object completes, the thread switches back to being in application space (line 8).

This mechanism also produces two versions of functions based on one code base. Consequently, it naturally prevents a situation such as recursive calls that can lead to stack overflow, which many previous works took pain to handle [8, 19].

In all, together with the space switch action in the wrappers, space separation is fully established, enabling replay to be confined to execution flow from the application only.

4.2 Memory Isolation

Space separation also allows us to establish memory isolation, a critical step to remove memory non-determinism mentioned in Section 2. Our goal is to make sure that memory usage in application space is deterministic between the logged run and the replayed run.

In most programs, memory blocks are located in the global data area, stack, as well as heap. Since the global data used by application and the replay tool have no intersections, there is nothing extra we need to do. However, this is not the case for stack and heap memories: memory allocation and release from both the application and replay tool can interleave in arbitrary ways. We employ a dedicated heap manager for application space that takes memory requests when a thread is in application space. This heap manager has no random factor, so that the same memory request sequence will produce the same memory footprints. The requests include the allocation of thread stack. Consequently, thread created from application space will inherently have a stack allocated from application space heap. In addition, we also allocate a stack from system space heap for these application threads. This way, when these threads invoke an intercepted function, the execution will be switched to system stack. This prevents our replay tool from being destroyed by bugs like buffer overflow from target application.

5. Event-based Replay

Our replay tool is *event-based*. More specifically, the execution of a thread is viewed as a succession of three types of events. The *API* events are the invocation of intercepted API functions, and they segment the thread execution into *continuation* events. Some of these API functions can take callback routines that will be executed

at some future points, and their invocations are the *callback* events. A multi-threaded/distributed application is a collection of these events from all threads running on all the machines. The task of logging is to 1) number these events and 2) record the outputs of API events, such that replay can process these events in increasing order while feeding the outputs of API events from the log. This way, the internal state of the application can be faithfully recreated as dictated by the application logic.

We number the events by assigning each event a 64-bit integer that we call logical clock. In the following, we will describe how clocks are assigned within a multi-threaded process, and then across processes in a distributed system. For the sake of clarity, these discussions assume that all API events are synchronous, and the asynchronous ones are discussed later, including both asynchronous I/O and exceptional control flow.

5.1 Assigning Clock within a Process

The first approach we adopt is rather conventional. We have developed a customized scheduler, which defines scheduling points at the boundary of intercepted API functions that are blocking. Then, the scheduler gives away n tokens, and let the runnable threads compete; a thread owns a token runs until the scheduling point. It is easy to see that changing n effectively sizes the number of concurrent running threads in the system. When n is one, all threads are sequentialized as they are being cooperatively scheduled. In this setting, assigning clock is trivial: the clock of each event increases monotonously.

This approach is safe because it eliminates non-determinism caused by data race (i.e. shared-memory access that arises outside the scope of synchronization). However, it has two limitations. First, the application should not have tight loop where the termination depends on a global variable that is only set by another thread. In other words, continuation event should have bounded termination time since our current implementation of the scheduler is not preemptive. Second, it decreases concurrency even when there are multiple processors. This is a more severe problem because deployed applications will always run on multiple processors, if present.

Our second approach respects the full concurrency but requires that the application not to have data race, a condition that is met by several distributed systems developed in parallel in our team and, we believe, by more and more well-disciplined systems. Each thread starts with the clock inherited from its creator (except the main thread which starts at zero), and the clock is increased in the same way independent of other threads if there are no dependencies to other threads. However, the clocks of the threads are modified to observe the happens-before relationships among events. This is done by capturing causality among API events that access the same resources. Resources here refer to synchronization objects, files and sockets, etc. We allocate a shadow memory block behind each resource handle to store, among others, the thread ID and the logical clock of the last API event that accesses the resource. When an API event involves a resource, its clock is updated with the maximum of its own clock and the clock value recorded on the memory block, and is incremented by one. This new clock value is stored into the shadow memory block when the API event completes. This way, the happens-before relationship is strictly reflected by the clock values of the events. Replay therefore process events in order, breaking ties with thread ID when events bear the same clock value.

A nice side effect is that we can track causalities and profile how long a thread is blocked on certain resources. Figure 2 illustrates the casualties we captured in a process of a distributed system we are building.

For applications that do have data race, we inherit the idea from RecPlay [16], which detects data race at replay time at a higher cost. We improve by only detecting data races for concurrent events. This approach reduces the cost of memory access tracking. We are now implementing this approach, and our preliminary experiment using Phoenix [3] shows that the slowdown for detecting data race at replay time is approximately 2$\sim$3 times for an I/O intensive application such as Wget.

5.2 Assign Clock Values Across Processes

We assume processes communicate with one another with messages through sockets, a model that is adopted by many distributed systems. We use Layered Service Provider (LSP) [2] to build a filter and message-processing layer and register the layer to Windows network service subsystem. All socket-based messages will go through this layer, and we embed the clock value of the sending socket in the outgoing message, and extract it at the receiving end. This clock then updates the clock of the receiving socket in the same way as described earlier. The engineering details are similar to [8]. For datagram protocols like UDP, we simply piggyback the clock value on each packet. For byte stream protocols like TCP, we add extra bytes to the byte stream in order to maintain message boundaries, since the receiver does not consume the incoming bytes along with the sending boundaries. This whole process is completely transparent to applications.

5.3 Asynchronous I/O

Asynchronous I/O is essential for high performance because it overlaps CPU processing with I/O, and is present in many modern operating systems, e.g. Windows, Mac OS, VMS, Linux 2.6. However, they bring two problems in replay, neither of which has been addressed in previous replay tools. We will illustrate these problems using asynchronous read. The first problem is that read buffer is not ready at the return point of the call. We must log the buffer after it is filled by the actual I/O operation and *before* it is modified by the application. Second, at replay time the call is not issued. Therefore, the operating system will never fill the buffer and there is no notification about the completion of the operation. We must find a way to present the logged data at the right point.

Our approach tracks the completion of the asynchronous read operation, recording both the buffer content as well as the logical clock, which is the point of time when we present the data during replay. This method depends on the OS notification mechanisms. Fortunately, there are only four such types on Windows; other platforms should be similar. We now detail how we handle the I/O completion port and event-based asynchronous I/O below.

I/O Completion Ports. When an asynchronous I/O is posted, an I/O completion port associates a pool of worker threads to the handle of the I/O object (e.g. file or socket) on which asynchronous I/O operations are issued [1]. When the I/O operation completes, an I/O completion packet will be queued to the port. The worker threads calls `GetQueuedCompletionStatus` to wait on the port, and wake up when I/Os are completed. The completion packet includes a `OVERLAPPED` that binds the issue and completion of the I/O operation, and this data structure traverses through the kernel.

Figure 3 illustrates how our replay tool tracks this whole process. Both `ReadFile` and `GetQueuedCompletionStatus` are converted into signal-slot structures. A local `OVERLAPPED` replaces the original one in `ReadFile`, and it has all necessary information including the pointer to the replaced `OVERLAPPED`. `GetQueuedCompletionStatus` is extended with a new slot after the native call, and the slot dereferences the `OVERLAPPED` structure, identify the buffer address and the length, and logs the output. Next, we recover the original `OVERLAPPED` and return. From the application's perspective, the call is handled with the full se-

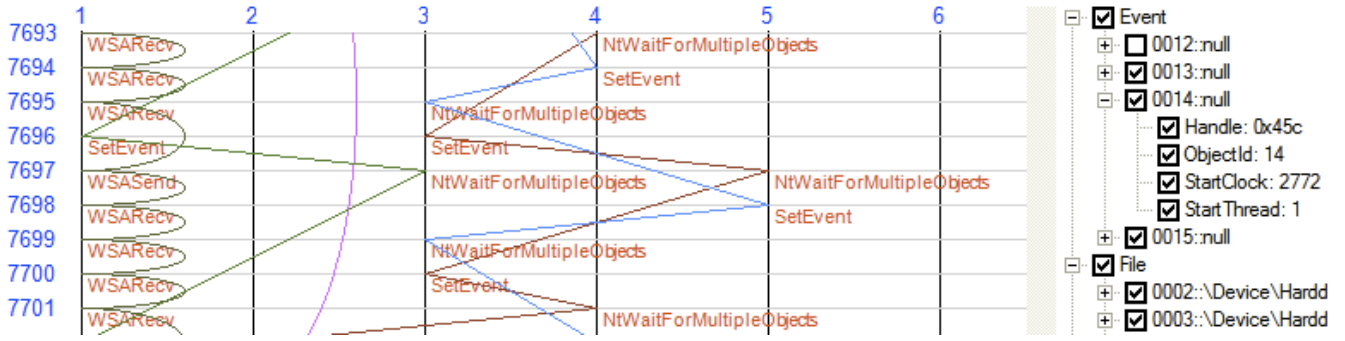

Figure 2. Causality visualizer. Here is an example of intra-process causalities in a multi-threaded distributed system. X axis presents the thread identities and Y axis indicates logical clock, and API events not related to resource are removed.

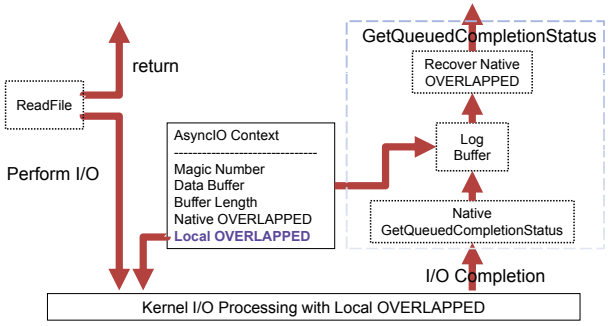

Figure 3. I/O Completion Port at log time

mantics intact. At replay time, the `ReadFile` is skipped, and the `GetQueuedCompletionStatus` call uses a slot to fill the read buffer, whose address and content are both kept in the log.

Event-based asynchronous I/O. Event is a synchronization object provided on Windows platform, which can be used to notify the completion of I/O operations. Application issues an asynchronous I/O operation with such an event embedded in the `OVERLAPPED` structure, and then waits on the event to be signaled later by the operating system once the operation completes. Before the native call is issued, we add a slot to replace the event with a new one, and then spawn a thread to wait on that event. When the operation completes, the new event is signaled. The dedicated thread wakes up and logs the buffer together with the logical clock, and then signals the native event. During replay, we feed the buffer at the logical clock recorded in the log.

5.4 Exceptional Control Flow

As we mentioned in Section 4, the virtual execution layer separates application space from system space, which is critical to eliminate the interference of the replay tool. Such separation is done normally at the entry and return point of intercepted API functions by setting thread specific flag. However, a noted exception is user functions that are invoked from system space, and we name them exceptional control flow. These functions are usually callbacks, and they are registered either explicitly or implicitly. Examples of the former include asynchronous `ReadFileEx` with callbacks, and message handler registration in many distributed systems, and the latter are per-thread dynamic-link library (DLL) initializations on Windows. While we designate a routine when we create a thread, the actual execution of the thread will first invoke entry points of in-process DLLs. This kind of callback registration is done automatically at run time by underlying operating system.

Our replay tool handles these control flows with callback events, and it deals with a number of issues. First, at replay time the intercepted API functions are not executed, thus triggering these callbacks need to be managed by the replay tool. Second, the callback functions are user routines, and they must be re-executed during replay in application space. However, during the logging phase they are triggered by the OS or the underlying libraries, and thus with respect to our virtual execution layer they are invoked when control resides in system space. We therefore wrap these callbacks and explicitly set flags for them, enforcing their execution in application space. We also record the logical clocks and parameters of the callback invocation during logging, and then explicitly invoke their executions by injecting a callback event when replaying.

Another problem is exception handling, whose handlers are registered at compile time. We assume that the underlying system space is trusted and any execution will not throw exceptions into these handlers. Under this assumption, the exception is only triggered in application space, and so are the exception handlers. The invocations of these exception handlers do not cross space boundary, hence we do not need to do anything.

6. Annotation-aware Code Generation

All the intercepted API functions need to be wrapped and converted to SignalEx structures as described in 4.1. Hand-coding of these wrappers are tedious and error-prone. Modern operating systems usually have a rich set of system functions: there are more than 1,000 functions in libc and even more in Win32. Instead of directly implementing the wrappers on each API, we annotate their prototypes appropriately and automatically generate the code. This section describes the implementation details.

6.1 Implementation

Most Windows functions are well annotated in the Standard Annotation Language (SAL) [9] in recent Windows Platform SDK. These annotations mostly focus on pointers, such as in/out on parameters, a buffer and its paired length specification. They concisely describe various aspects of attributes of functions and their parameters and thus form the base for our annotation-aware code generation. In the example below, `in` indicates that parameters `s`, `len` and `flags` are input parameters; `out` indicates that `buf` is the output buffer that must be logged for replay, with `bcount` to specify its initial length is `len` and the result length is the return value.

```
int recv(
    __in SOCKET s,
    __out_bcount_part(len, return) char * buf,
    __in int len,
    __in int flags
```

Annotation	Indication	Action
`xpointer`	allocated in system space	make a shadow copy
`xpointer(tls)`	allocated in system space per-thread internal buffer	copy to a per-thread internal buffer
`xpointer(global)`	allocated in system space global internal buffer	copy to a global internal buffer
`sched`	blocked	cooperatively schedule
`handle`	resource handle	track causalities of related API functions
`succ`	success condition	log only when the condition is met

Table 2. Extended Annotations.

```
);
```

Our parser reads annotation information, and then various scripting back-ends generate code automatically. First, a script generates code to convert each function into a SignalEx structure (Section 4), which can take slots as further extensions. Various other scripts are targeted to generate slots for different functions. For example, a script for logging generates a slot after each API function to record the contents of all output parameters, and another script for replay simply replaces the slot referencing original call with a slot feeding back recorded contents of output parameters.

Data exchanging across spaces needs special attention. For most API functions, it is the caller's responsibility to prepare and manage the buffer, and the API function just needs to fill the buffer. Thus, we only need to copy the content of the buffer to the log. On the contrary, some other functions return a buffer that is allocated internally. For example, `inet_ntoa` returns a string for a given network address, in which the buffer is maintained by the callee rather than the caller. This presents a few challenges. First, it is generally unsafe to re-execute these API functions at the replay time and hope that their outputs matches exactly with the one at the original run. Second, even if the content of the execution is reproducible, the call can return a pointer that may differ from the original run. If an application ever uses this pointer as an input, state corruption would inevitably happen. We solve this problem by allocating a shadow copy that is returned to the application at both logging and replay time, avoiding potential non-determinism caused by these API functions. The annotation `xpointer` instructs the script to generate appropriate codes. Specifically:

- An API function may allocate a new buffer in each call; meanwhile it must have a paired "free" API function, e.g. `getaddrinfo` and `freeaddrinfo`, `GetEnvironmentStrings` and `FreeEnvironmentStrings`. We use `xpointer` to specify such buffers.
- An API function returns a global/per-thread internal buffer; usually the returned buffer is guaranteed to be valid only until next corresponding calls, e.g. `GetCommandLine` and `inet_ntoa`. We use `xpointer(global)`/`xpointer(tls)` to indicate that the buffer is taken from a global/per-thread internal buffer space.

Table 2 summarizes the annotations we used, on top of what is already provided by SAL. The annotation `handle` specifies the resource used in an API to enable causality tracking, and the annotation `sched` specifies a blocking API as a scheduling point so that a slot for customized scheduler can be plugged in. Finally, the annotation `succ` tells the success predicate on functions, thus the logger ignores output parameters on functions that fail to satisfy their `succ` conditions to reduce the log time and size. It turns out that this needs to be carefully handled: socket API functions usually succeed when the return value equals zero; most traditional Win32 API functions succeed when the return value is not zero.

6.2 Corner Cases

Pointers to structure types must be handled carefully. For example, `getaddrinfo` is a socket API function that returns the head of an address list for a specified host name. The logger must traverse the entire linked-list to log the full information to restore it later, and this cannot be annotated by simply specifying the length of a buffer. This problem is similar to marshaling irregular data types when generating proxy/stub code for RPC. We could have solved this problem by adding more annotations to describe the layout of buffers. Instead, we enable customized logging and replay irregular data types via operator overloading on streams, which is a typical C++ idiom.

Only three API wrappers are hand-coded. Two API functions `RtlEnterCriticalSection` and `RtlLeaveCriticalSection` are hand-coded for efficiency. We also hand-code the socket function `WSAIoctl`, because it is too complicated and not general enough to worth the trouble of developing the script logic.

Overall, our experience is that automatic code generation is an essential and indispensable mechanism to build such a replay tool for realistic applications, which relies on a huge number of API functions.

7. Evaluation

This section presents quantitative measurements about major aspects of our replay tool. All experiments were performed on machines with 2.8 GHz Pentium 4 CPU, 512 MB memory and 100Mbps network card.

7.1 Code Generation

Table 3 gives the statistics of the functions intercepted to replay the applications listed in Section 7.4. These functions are spread within five different modules. As we proceed to support more applications, we will continue to intercept more functions.

There are a total of 751 API functions being intercepted. The replay column shows those that are both logged and replayed. The rest 94 are re-executed during replay because they are mostly console outputs; an example is `WriteConsole`. These API functions are a good example of hiding low-level complexities since they will invoke LPC calls. Intercepting these API functions also makes sure that the state involved is in system space, and thus will not produce any side-effects on the application state. The table also lists the number of functions that are customized, annotated to be asynchronous, scheduled, tracked for casualties (`handle`), with `succ`, with `xpointer`, with `xpointer(global)`, and with `xpointer(tls)`. The last four columns also show the average number of parameters, and among them that are annotated with `in`, `out`, and `optional`. Overall, our script generates 37,000 lines of code.

7.2 Wrapper

For every intercepted function, a wrapper takes charge of dispatching the execution into the correct subspace and signal-slot processing, as shown in Section 4.1. The wrapper is clearly on the critical

Modules	Wrap	Replay	Custed	async	sched	handle	succ	xpointer	xgloabl	xtls	arg	in	out	opt
ntdll	62	35	2	6	19	39	67	0	0	0	4.34	3.58	0.87	1.03
kernel32	509	442	0	0	0	98	31	4	2	0	2.62	2.15	0.58	0.05
advapi32	78	78	0	0	0	6	0	0	0	0	4.19	3.29	1.18	0.82
ws2_32	95	95	1	5	0	51	53	2	0	0	3.31	2.76	0.88	0.12
mswsock	7	7	0	5	0	6	0	0	0	7	6.29	5.29	1.29	0.14
Total	751	657	3	16	19	200	151	6	2	7	3.05	2.50	0.71	0.22

Table 3. Statistics of code generation

path. On x86 platform, the wrapper and the signal-slots produce 38 and 67 instructions on average, respectively. Thus, its overhead is quite low.

7.3 Logger

Our logger is per thread, dual-buffer and uses asynchronous I/O. We measure its impact to application under different I/O scenarios. We build an application that sequentially reads a file with a controlled throughput from 5 MB/s to 60 MB/s. Figure 4 illustrates the results. The x axis indicates the assigned throughput and the y axis is what the application achieves. The linear line shows that the application, running without the tool, can linearly approach the throughput at 60 MB/s. If the logger dumps log files on a different disk, there is no noticeable effect until reaching 50 MB/s. However, if the log files share the same disk with the application file, the curve caps out around 25 MB/s. This is expected because the tool and the application are heavily competing for I/O bandwidth.

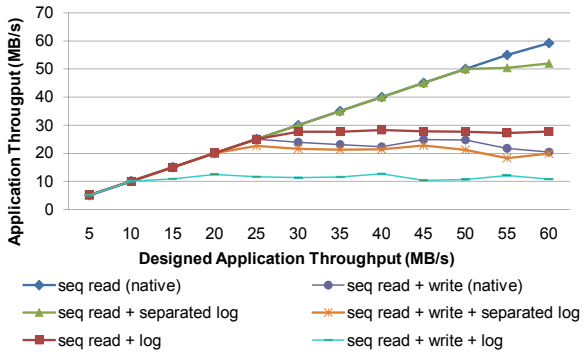

Figure 4. Performance of Logger

We add a thread in the application which writes a very big file at a constant throughput of 20MB/s. Note that these data do not need to be logged, but the write operation competes further for the bandwidth and randomizes disk access. We retest the three scenarios mentioned above with this trouble maker thread. The result, as presented in Figure 4 reaches the same conclusion. With a dedicated log disk, the overhead is negligible. Thus, for I/O intensive applications, a separate disk is required. For other application that has fewer I/O operations (e.g. below 10MB/s), sharing a disk should not cause noticeable performance degradation.

7.4 Applications

We have successfully replayed many large and complex applications that include several popular frameworks and libraries, including ADAPTIVE Communication Environment, Active Template Library, Apache Portable Runtime, Boost C++ libraries. Table 4 is an incomplete list. Amigo and PacificA are our two ongoing large-scale distributed systems; Amigo is a large-graph computation platform, and PacificA is a scalable semi-structured storage system.

We will present the overall performance results using MySQL, Apache HTTP server and libtorrent. Note that we did not evaluate

Category	Software Package
HTTP Server	Apache HTTP server, Null HTTPd
Database	Berkeley DB, MySQL, SQLite
Distributed System	Amigo, libtorrent, PacificA
Virtual Machine	Lua, Parrot, Python
Client	ApacheBench, cURL, PuTTY, Wget
Misc.	md5, OpenMP(VC), zip

Table 4. Software packages successfully replayed

the time for replay, since we believe the absolute performance is less interesting because the replay is an interactive process. However, our general observation is that the reply runs faster than the original run.

We installed the prebuilt binary version of MySQL Server 5.0.27 community version. [1] On the client side, we ran the standard MySQL benchmarks provided in the same package. Figure 5 presents the overall slowdown at log time for the test cases in the benchmark, and the average slowdown is 9%. The server side also produced a 6.46 GB log file (0.77 MB/s), based on which we successfully replayed MySQL server.

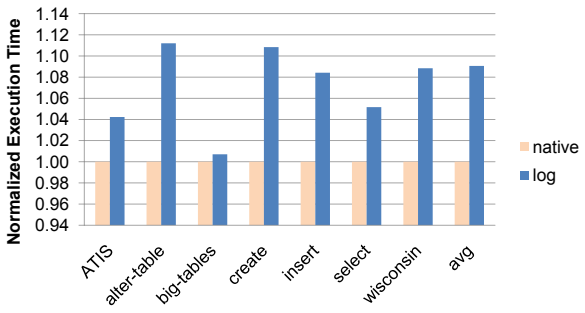

Figure 5. MySQL

We also experimented with the latest Apache Server 2.2.4, [2] and tested it with the default configuration (250 threads per child). We put HTML files with various sizes and downloaded them with the shipped ApacheBench in the same package. Figure 6 gives the overall performance, varying number of client concurrency and file size. For the particular case with 64KB file size and with concurrency level being 10, the test executes 100,000 times and generated 489 MB log files, with the average log throughput of 1.03 MB/s. We can see that larger download file leads to less slowdown. And, for the smallest size 16KB we tested, the average slowdown for all concurrency levels is 11.9%.

The logging overhead was a major concern of replay techniques, especially for bandwidth-intensive distributed systems like video streaming and file downloading applications [8]. To understand

[1] MySQL. `http://www.mysql.org/`

[2] Apache HTTP server. `http://httpd.apache.org/`

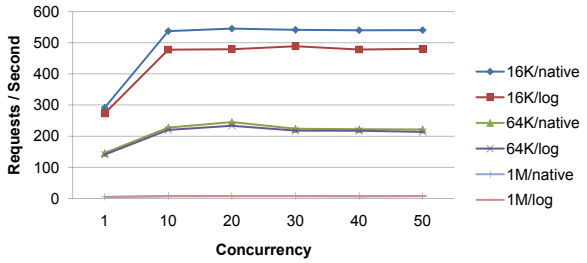

Figure 6. Apache

the performance with stressful bandwidth usage, we evaluated BitTorrent file downloading with unconstrained bandwidth in LAN. We built a C++ implementation named libtorrent 0.11[3] with Boost 1.33.1[4]. The experiment was conducted on 25 single-disk machines connected via 100Mbps LAN, with one seed and 24 downloaders. Figure presents the finish time distribution for downloading two files with file size 33.2MB and 1.15GB, respectively. The slowdown for the small file is quite small, ranging from 2% to 14%. For the 1.15GB file, the slowdown is from 56% to 110%. The resulting log size is over 6GB per client, and the average logging throughput is 6MB/s. Considering the random disk I/O caused by interference between libtorrent and logging, we believe the logging performance is reasonably good, and the slowdown is still acceptable. We further limited the uploading bandwidth by 1MB/s for each downloader, and found that the slowdown for downloading 1.15GB file was only 26%∼43%, on average 36%, which we think acceptable for logging and replaying many applications.

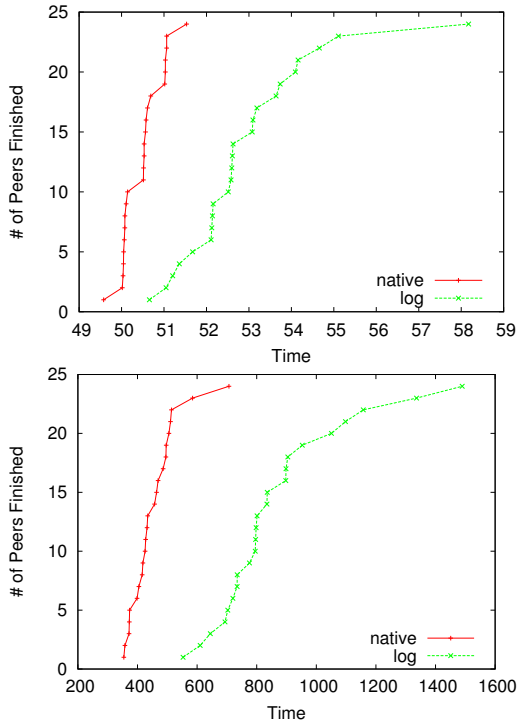

Figure 7. LibTorrent

Overall, we have successfully replayed many complex applications, including servers and distributed systems, those developed by our own and by the community. These experiments have validated our design, and gave us the confidence to move forward.

8. Discussion

Our early prototype was in fact quite similar to Jockey and liblog, in that we hand-code all the wrappers and perform log and replay of the complete process. During the process and after many trial-and-error scenarios, we gradually centered the tool on three key mechanisms: space separation, event-based logging and replay and annotation-aware code generation. We share some of our lessons below, and then discuss the limitation in the current implementation.

8.1 Experience

We realized early on that there needs to be a clean separation of the replayed application from the tool itself. The alternative we tried is to convert all API calls to RPCs, which are tunneled to a proxy process in which calls are executed and logged. We implemented an echo server with blocking socket API functions. However, supporting asynchronous calls under this model becomes extremely complicated. In addition, this model does not remove the anonymous threads that runs within the application process. This experience taught us that we should retain the in-process model, but establish isolation, which we did using thread specific flag to switch between and distinguish the two subspaces.

However, setting appropriate flags is not as straightforward as we thought. When the replay tool takes over the execution, we initially assume that all present threads live in application space. Unfortunately, while the main application thread indeed belongs to application space, the anonymous threads also creep into application space. These anonymous threads do not belong to any application logic and thus should not be logged and replayed. Further more, they may also cause unpredictable behavior. Thus, counter-intuitively, we set all except the main application thread to be in system space. As we described in Section 5, care must also be taken to handle exceptional control flows such as callbacks that traverse the subspaces.

Needless to say, our code generator relies on the correct annotation of API prototypes to begin with. It turns out that the Windows platform SDK header files still contain a number of bugs. The following is an example.

```
DWORD GetLongPathNameW (
  __in LPCWSTR lpszShortPath,
  __out_ecount_part(cchBuffer, return + 1)
       LPWSTR lpszLongPath,
  __in DWORD cchBuffer
);
```

The parameter `lpszLongPath` is not marked `optional`, which means it will not be null, and our logger can thus operate on the memory block at this address. However, it turns out that sometimes it can be, and this bug did crash the Wget.

Another lesson is that we need to identify and adjust the right layer of interception. We originally concentrated on the narrower system call layer. However, we later found that many system calls, such as `NtDeviceIoControlFile` and `NtReplyWaitReplyPort` for LPC, are difficult to be logged and replayed because of their complicated contracts. Therefore, we decided to move the layer to a higher level so as to hide the complexity. We scan the import table to make sure that an application, if it is to be logged and replayed, does not have calls that go directly to these low-level API functions. Wrapping higher layer API functions is what inspires the automatic code generation, since the surface is much wider.

We gained many positive lessons using the tool. An interesting application is to debug our own tool. If the tool crashes, we can al-

[3] Libtorrent. `http://libtorrent.sourceforge.net/`

[4] Boost C++ libraries. `http://www.boost.org/`

ways know at which clock the crash occurs. We then inject debugging event several steps before that clock, and replay to that point and inspect the code. Since replay is deterministic, this process can be repeatedly refined and has proved to be invaluable during the development of the tool.

This same approach is used to debug applications. For example, the example HTTP client in Parrot IR running on Parrot VM 0.4.8 [5] would terminate abnormally. We replayed to the clock just before the crash and quickly identified that Parrot VM called the C function `close` on a socket handle rather than the Win32 API `closesocket`, which was fine on Linux but illegal on Windows. A similar bug has been reported for Ruby [5]. We fixed the bug, and successfully ran and replayed the client.

8.2 Limitation in the Current Implementation

Our prototype has successfully replayed many complex applications. However, there are a still a number of issues that our future implementation plan to address. We believe that these limitations are not inherent to our methodologies, and can be lifted after further engineering using the same principles.

First of all, although we have intercepted and annotated more than 750 API calls, the intercepted API surface is still not wide enough. We will progressively expand this surface by replaying more applications.

Currently, we are able to replay interactive applications that use consoles. Windows messaging subsystem has a large body of protocols that need to be carefully examined before we can support GUI applications.

We have not handled `mmap` yet. In fact, `mmap` needs not be handled if the file is opened read-only, as is the example of the Apache test case. If the file is to be modified, all it takes is to make a shadow copy after the file is first opened and before it is modified. Since we have intercepted all file I/O calls, such modification is straightforward to implement.

Finally, as we mentioned earlier, we are implementing a lightweight checkpoint mechanism to target the always-on scenario.

9. Conclusion

In this paper, we propose three mechanisms for building a robust library-based replay tool. They are built around a virtual execution layer in the user space of the operating system. We use space separation with respect to this layer to redefine what should be replayed, and systematically reduce the complexity of handling issues such as in-address-tool interference. We decompose the execution of application code into various events based on our event view of systems. The event view enables event-based replay as well as support for many advanced features in modern operating systems such as asynchronous I/O. We adopt annotation-aware code generation to deal with large-scale interception and code generation for log and replay; this approach also allows the flexibility to redefine interception layer to hide the complexity of low-level mechanism. Our experiments have validated these methodologies, and we have successfully replayed many large and complex systems.

References

[1] I/O completion ports. `http://msdn2.microsoft.com/en-us/library/aa365198.aspx`.

[2] Layered service provider. `http://msdn2.microsoft.com/en-us/library/aa925739.aspx`.

[3] Phoenix compiler framework. `http://research.microsoft.com/phoenix/`.

[4] Purify. `http://www.ibm.com/software/awdtools/purify/`.

[5] Ruby bugs #2131. `http://rubyforge.org/tracker/index.php?func=detail&aid=2131&group_id=426&atid=1698`.

[6] Signals and slots. `http://doc.trolltech.com/signalsandslots.html`.

[7] 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 *USENIX Symposium on Operating System Design and Implementation*, 2002.

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

[9] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *International Conference on Software Engineering*, 2006.

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

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

[12] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *International Parallel and Distributed Processing Symposium*, 2000.

[13] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[14] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using Strata. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[15] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *International Symposium on Computer Architecture*, 2005.

[16] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. In *ACM Transactions on Computer Systems*, volume 17, pages 133–152, 1999.

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

[18] M. Ronsse and W. Zwaenepoel. Execution replay for treadmarks. In *EUROMICRO Workshop on Parallel and Distributed Processing*, 1997.

[19] Y. Saito. Jockey: A userspace library for record-replay debugging. In *International Workshop on Automated and Analysis-Driven Debugging*, 2005.

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

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

[22] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, 2003.

[5] Parrot virtual machine. `http://www.parrotcode.org/`