Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism

Dongyoon Lee Benjamin Wester Kaushik Veeraraghavan
Satish Narayanasamy Peter M. Chen Jason Flinn
Dept. of EECS, University of Michigan
{dongyoon,bwester,kaushikv,nsatish,pmchen,jflinn}@umich.edu

Abstract

Deterministic replay systems record and reproduce the execution of a hardware or software system. While it is well known how to replay uniprocessor systems, replaying shared memory multiprocessor systems at low overhead on commodity hardware is still an open problem. This paper presents Respec, a new way to support deterministic replay of shared memory multithreaded programs on commodity multiprocessor hardware. Respec targets *online* replay in which the recorded and replayed processes execute concurrently.

Respec uses two strategies to reduce overhead while still ensuring correctness: speculative logging and externally deterministic replay. Speculative logging optimistically logs less information about shared memory dependencies than is needed to guarantee deterministic replay, then recovers and retries if the replayed process diverges from the recorded process. Externally deterministic replay relaxes the degree to which the two executions must match by requiring only their system output and final program states match. We show that the combination of these two techniques results in low recording and replay overhead for the common case of data-race-free execution intervals and still ensures correct replay for execution intervals that have data races.

We modified the Linux kernel to implement our techniques. Our software system adds on average about 18% overhead to the execution time for recording *and* replaying programs with two threads and 55% overhead for programs with four threads.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability; D.4.6 [*Operating Systems*]: Security and Protection; D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Performance, Reliability

Keywords Replay, Speculative execution, External determinism

1. Introduction

Deterministic replay systems are used to record and reproduce the execution of a hardware or software system. This ability can be used to improve systems along many dimensions, including reliability, security, and debuggability. For example, deterministic replay is an efficient way to keep the state of a backup synchronized with the state of a primary machine [7]; it can be used to parallelize or offload heavyweight analysis from production machines [9, 28]; it can be combined with minor perturbations to diagnose or avoid faults [33, 38]; it can enable detailed analysis for forensics [10] or computer architecture research [24, 42]; and it can provide the illusion of reverse execution and time-travel debugging [16, 36].

The general idea behind deterministic replay is to log all non-deterministic events during recording and reproduce these events during replay. Deterministic replay for uniprocessors can be provided at low overhead because non-deterministic events occur at relatively low frequencies (e.g., interrupts or data from input devices and clocks), so logging them adds relatively little overhead.

Unfortunately, it is much harder to provide deterministic replay of shared memory multithreaded programs on multiprocessors because shared memory accesses add a high-frequency source of non-determinism. A variety of approaches have been proposed to reproduce this non-determinism by logging a precise order of shared memory accesses, but these approaches either require custom hardware [15, 21, 23, 40], or are prohibitively slow for many parallel applications [11]. Other software approaches that target efficiency guarantee determinism only for race-free programs [30, 34] or only reproduce the partial state of the original system [32].

This paper describes Respec, a new way to support deterministic replay of a shared memory multithreaded program execution on a commodity multiprocessor. Respec's goal is to provide fast execution in the common case of data-race-free execution intervals and still ensure correct replay for execution intervals with data races (albeit with additional performance cost). Respec targets *online* replay in which the recorded and replayed processes execute concurrently.

Respec is based on two insights. First, Respec can optimistically log the order of memory operations less precisely than the level needed to guarantee deterministic replay, while executing the recorded execution speculatively to guarantee safety. After a configurable number of misspeculations (that is, when the information logged is not enough to ensure deterministic replay for an interval), Respec rolls back execution to the beginning of the current interval and re-executes with a more precise logger. Second, Respec can detect a misspeculation for an interval by concurrently replaying the recorded interval on spare cores and checking if its system output and final program states (architectural registers and memory state) matches those of the recorded execution. We argue in Section 2.1 that matching the system output and final program states of the two executions is sufficient for most applications of replay.

Respec works in the following four phases:

First, Respec logs most common, but not all, synchronization operations (e.g., lock and unlock) executed by a shared memory multi-

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

threaded program. Logging and replaying the order of all synchronization operations guarantees deterministic replay for the data-race-free portion of programs [34], which is usually the vast majority of program execution.

Second, Respec detects when logging synchronization operations is insufficient to reproduce an interval of the original run. Respec concurrently replays a recorded interval on spare cores and compares it with the original execution. Since Respec's goal is to reproduce the visible output and final program states of the original execution, Respec considers any deviation in system call output or program state at the end of an interval to be a failed replay. Respec permits the original and replayed execution to diverge during an interval, as long as their system output and the program memory and register states converge by the end of that interval.

Third, Respec uses speculative execution to hide the effects of failed replay intervals and to transparently rollback both recorded and replayed executions. Respec uses operating system speculation [27] to defer or block all visible effects of both recorded and replayed executions until it verifies that these two executions match.

Fourth, after rollback, Respec retries the failed interval of execution by serializing the threads and logging the schedule order, which guarantees that the replay will succeed for that interval.

Our results show that Respec shared memory multiprocessor record and replay is efficient. For a combination of PARSEC and SPLASH-2 benchmarks, as well as the pbzip2, pfscan, aget and Apache applications, Respec adds only an average 18% overhead to execution time when two threads are replayed and 55% when four threads are replayed.

2. Replay guarantees

Replay systems provide varying guarantees. This section discusses two types of guarantees that are relevant to Respec: fidelity level and online versus offline replay.

2.1 Fidelity level

Replay systems differ in their fidelity of replay and the resulting cost of providing this fidelity. One example of differing fidelities is the abstraction level at which replay is defined. Prior machine-level replay systems reproduce the sequence of instructions executed by the processor and consequently reproduce the program state (architectural registers and memory state) of executing programs [7, 10, 41]. Deterministic replay can also be provided at higher levels of a system, such as a Java virtual machine [8] or a Unix process [36], or lower levels of a system, such as cycle accuracy for interconnected components of a computer [35]. Since replay is deterministic only above the replayed abstraction level, lower-level replay systems have a greater scope of fidelity than higher-level replay systems.

Multiprocessor replay adds another dimension to fidelity: how should the replaying execution reproduces the interleaving of instructions from different threads. No proposed application of replay requires the exact time based ordering of all instructions to be reproduced. Instead, one could reproduce data from shared memory reads, which, when combined with the information recorded for uniprocessor deterministic replay, guarantees that each thread executes the same sequence of instructions. Reproducing data read from shared memory can be implemented in many ways, such as reproducing the order of reads and writes to the same memory location, or logging the data returned by shared memory reads.

Replaying the order of dependent shared memory operations is sufficient to reproduce the execution of each thread. However, for most applications, this degree of fidelity is exceedingly difficult to provide with low overhead on commodity hardware. Logging the order or results of critical shared memory operations is sufficient but costly [11].

Logging higher-level synchronization operations is sufficient to replay applications that are race-free with respect to those synchronization operations [34]. However, this approach does not work for programs with data races. In addition, for legacy applications, it is exceedingly difficult to instrument *all* synchronization operations. Such applications may contain hundreds or thousands of synchronization points that include not just Posix locks but also spin locks and lock-free waits that synchronize on shared memory values. Further, the libraries with which such applications link contain a multitude of synchronization operations. GNU glibc alone contains over 585 synchronization points, counting just those that use atomic instructions. Instrumenting all these synchronization points, including those that use no atomic instructions, is difficult.

Further, without a way to correct replay divergence, it is incorrect to instrument only some of the synchronization points, assuming that uninstrumented points admit only benign data races. Unrelated application bugs can combine with seemingly benign races to cause a replay system to produce an output and execution behavior that does not match those of the recorded process. For instance, consider an application with a bug that causes a wild store. A seemingly benign data race in glibc's memory allocation routine may cause an important data structure to be allocated at different addresses. During a recording run, the structure is allocated at the same address as the wild store, leading to a crash. During the replay run, the structure is allocated at a different address, leading to an error-free execution. A replay system that allowed this divergent behavior would clearly be incorrect. To address this problem, one can take either a pessimistic approach, such as logging all synchronization operations or shared memory addresses, or an optimistic approach, such as the rollback-recovery Respec uses to ensure that the bug either occurs in both the recorded and replayed runs or in neither.

The difficulty and inefficiency of pessimistic logging methods led us to explore a new fidelity level for replay, which we call *externally deterministic replay*. Externally deterministic replay guarantees that (1) the replayed execution is indistinguishable from the original execution *from the perspective of an outside observer*, and (2) the replayed execution is a *natural* execution of the target program, i.e., the changes to memory and I/O state are produced by the target program. The first criterion implies that the sequence of instructions executed during replay *cannot be proven to differ* from the sequence of instructions executed during the original run because all observable output of the two executions are the same. The second criterion implies that each state seen during the replay was able to be produced by the target program; i.e. the replayed execution must match the instruction-for-instruction execution of one of the possible executions of the unmodified target system that would have produced the observed states and output. Respec exploits these relaxed constraints to efficiently support replay that guarantees identical output and natural execution even in the presence of data races and unlogged synchronization points.

We assume an outside observer can see the output generated by the target system, such as output to an I/O device or to a process outside the control of the replay system. Thus, we require that the outputs of the original and replayed systems match. Reproducing this output is sufficient for many uses of replay. For example, when using replay for fail-stop fault tolerance [7], reproducing the output guarantees that the backup machine can transparently take over when the primary machine fails; the failover is transparent because the state of the backup is consistent with the sequence of output produced before the failure. For debugging [12, 16], this guarantees

that all observable symptoms of the bug are reproduced, such as incorrect output or program crashes (reproducing the exact timing of performance bugs is outside our scope of observation).

We also assume that an outside observer can see the final program state (memory and register contents) of the target system at the end of a replay interval, and thus we require that the program states of the original and replayed systems match at the end of each replay interval.

Reproducing the program state at the end of a replay interval is mandatory whenever the program states of both the recording and replaying systems are used. For example, when using replay for tolerating non-fail-stop faults (e.g., transient hardware faults), the system must periodically compare the state of the replicas to detect latent faults. With triple modular redundancy, this comparison allows one to bound the window over which at most one fault can occur. With dual modular redundancy and retry, this allows one to verify that a checkpoint has no latent bugs and therefore is a valid state from which to start the retry.

Another application of replay that requires the program states of the original and replayed systems to match is parallelizing security and reliability checks, as in Speck [28]. Speck splits an execution into multiple epochs and replays the epochs in parallel while supplementing them with additional checks. Since each epoch starts from the program state of the original run, the replay system must ensure that the final program states of the original and replayed executions match, otherwise the checked execution as a whole is not a natural, continuous run.

Note that externally deterministic replay allows a more relaxed implementation than prior definitions of deterministic replay. In particular, externally deterministic replay does not guarantee that the replayed sequence of instructions matches the original sequence of instructions, since this sequence of instructions is, after all, not directly observable. We leverage this freedom when we evaluate whether the replayed run matches the original run by comparing only the output via system calls and the final program state. Reducing the scope of comparison helps reduce the frequency of failed replay and subsequent rollback.

2.2 Online versus offline replay

Different uses of deterministic replay place different constraints on replay speed. For some uses, such as debugging [12, 16] or forensics [10], replay is performed after the original execution has completed. For these *offline* uses of replay, the replay system may execute much slower than the original execution [32].

For other uses of replay, such as fault tolerance and decoupled [9] or parallel checks [28], the replayed execution proceeds in parallel with the original execution. For these *online* uses of replay, the speed of replayed execution is important because it can limit the overall performance of the system. For example, to provide synchronous safety guarantees in fault tolerance or program checking, one cannot release output until the output is verified [19].

In addition to the speed of replay, online and offline scenarios differ in how often one needs to replay an execution. Repeated replay runs are common for offline uses like cyclic debugging, so these replay systems must guarantee that the replayed run can be reproduced at will. This is accomplished either by logging complete information during the original run [10], or by supplementing the original log during the first replayed execution [32]. In contrast, online uses of replay need only replay the run a fixed number of times (usually once).

Respec is designed for use in online scenarios. It seeks to minimize logging and replay overhead so that it can be used in production settings with synchronous guarantees of fault tolerance or program error checking. Respec guarantees that replay can be done any number times when a program is executing. If replay needs to be repeated offline, Respec could store the log in permanent storage. The recorded log would be sufficient for deterministically replaying race-free intervals offline. For offline replay of racy intervals, a replay search tool [1, 18, 32] could be used.

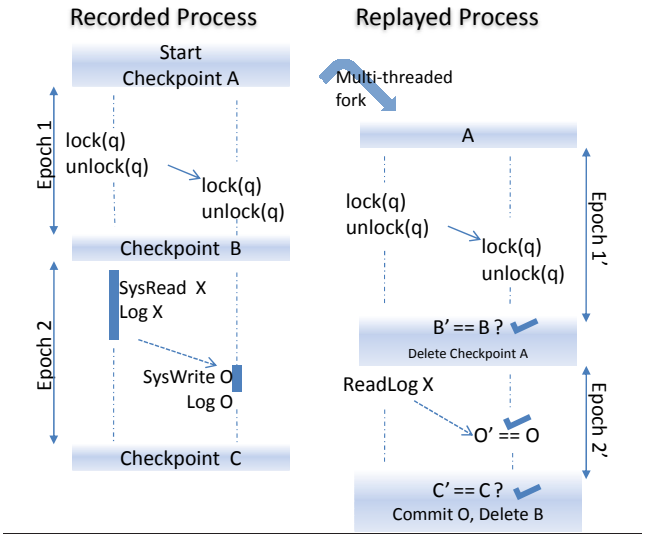

Figure 1. An execution in Respec with two epochs.

3. Design

This section presents the design of Respec, which supports online, externally deterministic replay of a multithreaded program execution on a multiprocessor.

3.1 Overview

Respec provides deterministic replay for one or more processes. It replays at the process abstraction by logging the results of system calls and low-level synchronization operations executed by the recording process and providing those logged results to the replayed process in lieu of re-executing the corresponding system calls and synchronization operations. Thus, kernel activity is not replayed.

Figure 1 shows how Respec records a process and replays it concurrently. At the start, the replayed process is forked off from the recorded process. The fork ensures deterministic reproduction of the initial state in the replayed process. Respec checkpoints the recording process at semi-regular intervals, called *epochs*. The replayed process starts and ends an epoch at exactly the same point in the execution as the recording process.

During an epoch, each recorded thread logs the input and output of its system calls. When a replayed thread encounters a system call, instead of executing it, it emulates the call by reading the log to produce return values and address space modifications identical to those seen by the recorded thread. To deterministically reproduce the dependencies between threads introduced by system calls, Respec records the total order of system call execution for the recorded process and forces the replayed process to execute the calls in the same order.

To reproduce non-deterministic shared memory dependencies, Respec optimistically logs just the common user-level synchronization operations in GNU glibc. Rather than enforcing a total order

over synchronization operations, Respec enforces a partial order by tracking the causal dependencies introduced by synchronization operations. The replayed process is forced to execute synchronization operations in an order that obeys the partial ordering observed for the recording process. Enforcing the recorded partial order for synchronization operations ensures that all shared memory accesses are ordered, provided the program is race free.

Replay, however, could fail when an epoch executes an unlogged synchronization or data race. Respec performs a *divergence check* to detect such replay failures. A naive divergence check that compares the states of the two executions after every instruction or detects unlogged races would be inefficient. Thus, Respec uses a faster check. It compares the arguments passed to system calls in the two executions and, at the end of each epoch, it verifies that the memory and resister state of the recording and replayed process match. If the two states agree, Respec commits the epoch, deletes the checkpoint for the prior epoch, and starts a new epoch by creating a new checkpoint. If the two states do not match, Respec rolls back recording and replayed process execution to the checkpoint at the beginning of the epoch and retries the execution. If replay again fails to produce matching states, Respec uses a more conservative logging scheme that guarantees forward progress for the problem epoch. Respec also rolls back execution if the synchronization operations executed by the replayed process diverge from those issued by the recorded process (e.g., if a replay thread executes a different operation than the one that was recorded) since it is unlikely that the program states will match at the end of an epoch.

Respec uses speculative execution implemented by Speculator [27] to support transparent application rollback. During an epoch, the recording process is prevented from committing any external output (e.g., writing to the console or network). Instead, its outputs are buffered in the kernel. Outputs buffered during an epoch are only externalized after the replayed process has finished replaying the epoch and the divergence check for the epoch succeeds.

3.2 Divergence Checks

Checking intermediate program state at the end of every epoch is not strictly necessary to guarantee externally deterministic replay. It would be sufficient to check just the external outputs during program execution. However, checking intermediate program state has three important advantages. First, it allows Respec to commit epochs and release system output. It would be unsafe to release the system output without matching the program states of the two processes. Because, it might be prohibitively difficult to reproduce the earlier output if the recorded and replayed processes diverge at some later point in time. For example, a program could contain many unlogged data races, and finding the exact memory order to reproduce the output could be prohibitively expensive. Second, intermediate program state checks reduce the amount of execution that must rolled back when a check fails. Third, they enable other applications such as fault tolerance, parallelizing reliability checks, etc., as discussed in Section 2. Though intermediate program state checks are useful, they incur an additional overhead proportional to the amount of memory modified by an application. Respec balances these tradeoffs by adaptively configuring the length of an epoch interval. It also reduces the cost of checks by parallelizing them and only comparing pages modified in an epoch.

Respec's divergence check is guaranteed to find all instances when the replay is not externally deterministic with respect to the recorded execution. But, this does not mean that execution of an unlogged race will always cause the divergence check to fail. For several types of unlogged races, Respec divergence check will succeed. This reduces the number of rollbacks necessary to produce an externally deterministic replay.

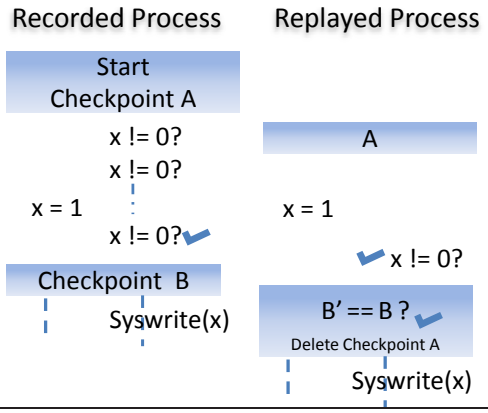

Figure 2. A race that produces the same program state irrespective of the order between the racing memory operations. Although the number of reads executed by the replayed process is different from the recorded process causing a transient divergence, the executions eventually converge to the same program state.

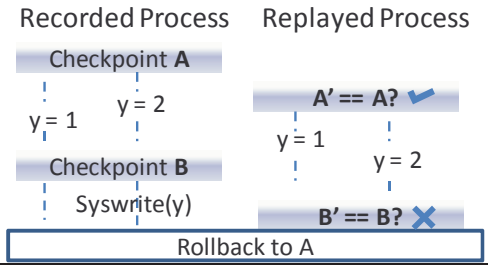

Figure 3. An execution with a data race that causes the replayed process to produce a memory state different from that of the recorded process. The divergence check fails and the two processes are rolled back to an earlier checkpoint.

First, the replayed process might produce the same causal relationship between the racing operations as in the recorded execution. Given that Respec logs a more conservative order between threads (a total order for system calls and even the partial order recorded for synchronization operations is stricter than necessary as discussed in Section 4.3.1), the replayed process is more likely to reproduce the same memory order.

Second, two racing memory operations might produce the same program state, either immediately or sometime in future, irrespective of the order of their execution. This is likely if the unlogged race is a synchronization race or a benign data race [25]. For example, two racing writes could be writing the same value, the write in a read-write race could be a silent write, etc. Another possibility is that the program states in the two processes might converge after a transient divergence without affecting system output. Note that a longer epoch interval would be beneficial for such cases, as it increases the probability of checking a converged program state.

Figure 2 shows an epoch with an unlogged synchronization race that does not cause a divergence check to fail. The second thread waits by iterating in a spin loop until the first thread sets the variable `x`. Because there is no synchronization operation that orders the write and the reads, the replayed process might execute a different number of reads than the recorded process. However, the program states of the replayed and recorded processes eventually converge, and both processes would produce the same output for any later system call dependent on `x`. Thus, no rollback is triggered.

However, for harmful races that should happen rarely, Respec's divergence check could trigger a rollback. Figure 3 shows an epoch with a harmful data race where the writes to a shared variable `y` are not ordered by any logged synchronization operation. The replayed execution produces a memory state different from the recorded execution. This causes the divergence check to fail and initiate a recovery process. This example also shows why it is important to check the intermediate program states before committing an epoch. If we commit an epoch without matching the program states, the two executions would always produce different output at the system call following the epoch. Yet, the replay system could not roll back past the point where the executions diverged in order to retry and produce an externally deterministic replay.

4. Implementation

4.1 Checkpoint and multithreaded fork

Rollback and recovery implementations often use the Unix copy-on-write `fork` primitive to create checkpoints efficiently [13, 27]. However, Linux's `fork` works poorly for checkpointing multithreaded processes because it creates a child process with only a single thread of control (the one that invoked `fork`). We therefore created a new Linux primitive, called a *multithreaded fork*, that creates a child process with the same number of threads as its parent.

Not all thread states are safe to checkpoint. In particular, a thread cannot be checkpointed while executing an arbitrary kernel routine because of the likelihood of violating kernel invariants if the checkpoint is restored (this is possible because kernel memory is not part of the checkpoint). For example, the restarted thread would need to reacquire any kernel locks held prior to the checkpoint since the original checkpointed process would release those locks. It would also need to maintain data invariants; e.g., by not incrementing a reference count already incremented by the original process, etc.

Consequently, Respec only checkpoints a thread when it is executing at a known safe point: kernel entry, kernel exit, or certain interruptible sleeps in the kernel that we have determined to be safe. The thread that initiates a multithreaded fork creates a barrier on which it waits until all other threads reach a safe point. Once all threads reach the barrier, the original thread creates the checkpoint, then lets the other threads continue execution. For each thread, the multithreaded fork primitive copies the registers pushed onto the kernel stack during kernel entry, as well as any thread-level storage pointers. The address space is duplicated using `fork`'s copy-on-write implementation.

Respec uses the multithreaded fork primitive in two circumstances: first, to create a replayed process identical to the one being recorded, and second, to create checkpoints of the recorded process that may later be restored on rollback. In the first case, the recorded process simply calls the multithreaded fork primitive directly. In the second case, the checkpointing code also saves additional information that is not copied by `fork` such as the state of the file descriptors and pending signals for the child process. The child process is not put on the scheduler's run queue unless the checkpoint is restored; thus, unless a rollback occurs, the child is merely a vessel for storing state. Respec deletes checkpoints once a following checkpoint has been verified to match for the recorded and replayed processes.

Respec checkpoints the recorded process at semi-regular intervals, called *epochs*. It takes an initial checkpoint when the replayed process is first created. Then, it waits for a predetermined amount of time (the epoch interval) to pass. After the epoch interval elapses, the next system call by any recorded thread triggers a checkpoint. After the remaining threads reach the multithreaded fork barrier and the checkpoint is created, all threads continue execution. The recorded process may execute several epochs ahead of the replayed process. It continues until either it is rolled back (due to a failed divergence check) or its execution ends.

Respec sets the epoch interval adaptively. There are two reasons to take a checkpoint. First, a new checkpoint bounds the amount of work that must be redone on rollback. Thus, the frequency of rollback should influence the epoch interval. Respec initially sets the epoch interval to a maximum value of one second. If a rollback occurs, the interval is reduced to 50 ms. Each successful checkpoint commit increases the epoch interval by 50 ms until the interval reaches its maximum value. The second reason for taking a checkpoint is to externalize output buffered during the prior epoch (once the checkpoint is verified by comparing memory states of the recorded and replayed process). To provide acceptable latency for interactive tasks, Respec uses output-triggered commits [29] to receive a callback when output that depends on a checkpoint is buffered. Whenever output occurs during an epoch, we reduce that epoch's interval to 50 ms. If the epoch has already executed for longer than 50 ms, a checkpoint is initiated immediately. Note that the actual execution time of an epoch may be longer than the epoch interval due to our barrier implementation; a checkpoint cannot be taken until all threads reach the barrier.

4.2 Speculative execution

The recorded process is not allowed to externalize output (e.g., send a network packet, write to the console, etc.) until both the recorded and replayed processes complete the epoch during which the output was attempted and the states of the two processes match. A conservative approach that meets this goal would block the recorded process when it attempts an external output, end the current epoch, and wait for the replayed process to finish the epoch. Then, if the process states matched, the output could be released. This approach is correct, but can hurt performance by forcing the recorded and replayed process to execute in lockstep.

A better approach is available given operating system support for speculative execution. One can instead execute the recorded thread speculatively and either buffer the external output (if it is asynchronous) or allow speculative state to propagate beyond the recorded process as long as the OS guarantees that the speculative state can be rolled back and that speculative state will not causally effect any external output. We use Speculator [27] to do just that.

In particular, Speculator allows speculative state to propagate via fork, file system operations, pipes, Unix sockets, signals, and other forms of IPC. Thus, additional kernel data structures such as files, other processes, and signals may themselves become speculative without blocking the recorded process. External output is buffered within the kernel when possible and only released when the checkpoints on which the output depends are committed. External inputs such as network messages are saved as part of the checkpoint state so that they can be restored after a rollback. If propagation of speculative state or buffering of output is not possible (e.g., if the recorded thread makes an RPC to a remote server), the recorded thread ends the current epoch, blocks until the replayed thread catches up and compares states, begins a new epoch, and releases the output. We currently use this approach to force an epoch creation on all network operations, which ensures that an external computer never sees speculative state. Respec allows multiple pairs of processes to be recorded and replayed independently, with the exception that two processes that write-share memory must be recorded and replayed together.

4.3 Logging and replay

Once a replayed process is created, it executes concurrently with its recorded process. Each recorded thread logs the system calls and user-level synchronization operations that it performs, while the corresponding replayed thread consumes the records to recreate the results and partial order of execution for the logged operations.

Conceptually, there is one logical log for each pair of recorded and replayed threads. Yet, for performance and security reasons, our implementation uses two physical logs: a log in kernel memory contains system call information and a user-level log contains user-level synchronization operations. If we logged both types of operations in the kernel's address space, then processes would need to enter kernel mode to record or replay user-level synchronization operations. This would introduce unacceptable overhead since most synchronization operations can be performed without system calls using a single atomic instruction. On the other hand, logging all operations in the application's address space would make it quite difficult to guarantee externally deterministic replay for a malicious application's execution. For instance, a malicious application could overwrite the results of a *write* system call in the log, which would compromise the replayed process's output check. By placing only the data necessary for performance at user-level, the verification of log records described in Section 4.7 is considerably simplified.

4.3.1 User-level logging

At user-level, we log the order of the most common low-level synchronization operations in glibc, such as locks, unlocks, futex waits, and futex wakes. The Posix thread implementation in glibc consists of higher-level synchronization primitives built on top of these lower-level operations. By logging only low-level operations, we reduce the number of modifications to glibc and limit the number of operation types that we log. Our implementation currently logs synchronization primitives in the Posix threads, memory allocation, and I/O components of glibc. An unlogged synchronization primitive in the rest of glibc, other libraries, or application code could cause the recorded and replayed processes to diverge. For such cases, we rely on rollback to re-synchronize the process states. As our results show, logging these most common low-level synchronization points is sufficient to make rollbacks rare in the applications we have tested.

However, for an application that heavily uses handcrafted synchronizations our approach might lead to frequent rollbacks. A simple solution would be to require programmers to annotate synchronization accesses so that we could instrument and log them. In fact, recently proposed Java [20] and C++0x [5] memory models already require programmers to explicitly annotate synchronization accesses using `volatile` and `atomic` keywords.

Respec logs the entry and exit of each synchronization operation. Each log record contains the type of operation, its result, and its partial order with respect to other logged operations. The partial order captures the total order of all the synchronization operations accessing the same synchronization variable and the program order of the synchronization operations executed in the same thread.

To record the partial order, we hash the address of the lock, futex, or other data structure being operated upon to one of a fixed number of global record clocks (currently 512). Each recorded operation atomically increments a clock and records the clock's value in a producer-consumer circular buffer shared between the recorded thread and its corresponding replayed thread. Thus, recording a log record requires at most two atomic operations (one to increment a clock and the other to coordinate access to the shared buffer). This allows us to achieve reasonable overhead even for synchronization operations that do not require a system call.

Using fewer clocks than the number of synchronization variables reduces the memory cost, and also produces a correct but stricter partial order than is necessary to faithfully replay a process. A stricter order is more likely to replay the correct order of racing operations and thereby reduce the number of rollbacks, as discussed in Section 3.2.

When a replayed thread reaches a logged synchronization operation, it reads the next log record from the buffer it shares with its recorded thread, blocking if necessary until the record is written. It hashes the logged address of the lock, futex, etc. to obtain a global replay clock and waits until the clock reaches the logged value before proceeding. It then increments the clock value by one and emulates the logged operation instead of replaying the synchronization function. It emulates the operation by modifying memory addresses with recorded result values as necessary and returning the value specified in the log. Each synchronization operation consumes two log records, one on entry and one on exit, which recreates the partial order of execution for synchronization operations.

Respec originally used only a single global clock to enforce a total order over all synchronization operations, but we found that this approach reduced replay performance by allowing insufficient parallelism among replayed threads. We found that the approach of hashing to a fixed number of clocks greatly increased replay performance (by up to a factor of 2–3), while having only a small memory footprint. Potentially, we could use a clock for each lock or futex, but our results to date have shown that increasing beyond 512 clocks offers only marginal benefits.

4.3.2 Kernel logging

Respec uses a similar strategy to log system calls in the kernel. On system call entry, a recorded thread logs the type of call and its arguments. For arguments that point to the application's address space, e.g., the buffer passed to `write`, Respec logs the values copied into the kernel during system call execution. On system call exit, Respec logs the call type, return value, and any values copied into the application address space. When a replayed thread makes a system call, it checks that the call type matches the next record in the log. It also verifies that the arguments to the system call match. It then reads the corresponding call exit record from its log, copies any logged values into the address space of the replayed process and returns the logged return value.

Respec currently uses a single clock to ensure that the recorded and replayed process follow the same total order for system call entrance and exit. This is conservative but correct. Enforcing a partial order is possible, but requires us to reason about the causal interactions between pairs of system calls; e.g., a file *write* should not be reordered before a *read* of the same data.

Using the above mechanism, the replayed process does not usually perform the recorded system call; it merely reproduces the call's results. However, certain system calls that affect the address space of the application must be re-executed by the calling process. When Respec sees log records for system calls such as *clone* and *exit*, it performs these system calls to create or delete threads on behalf of the replayed process. Similarly, when it sees system calls that modify the application address space such as *mmap2* and *mprotect*, it executes these on behalf of the replayed process to keep its address space identical with that of the recorded process. This replay strategy does not recreate most kernel state associated with a replaying process (e.g., the file descriptor table), so a process cannot transition from replaying to live execution. To support such a transition, the kernel could deterministically re-execute native system calls [10] or virtualized system calls [31].

When the replayed process does not re-execute system calls, we do not need to worry about races that occur in the kernel code; the effect on the user-level address space of any data race that occurred in the recorded process will be recreated. For those system calls such as *mmap2* that are partially re-created, a kernel data race between system calls executed by different threads may lead to a divergence (e.g., different return values from the `mmap2` system call or a memory difference in the process address space). The divergence would trigger a rollback in the same manner as a user-level data race.

Because signal delivery is a source of non-determinism, Respec does not interrupt the application to deliver signals. Instead, signals are deferred until the next system call, so that they can be delivered at the same point of execution for the recorded and replayed threads. A data races between a signal handler and another thread is possible; such races are handled by Respec's rollback mechanism.

4.4 Detecting divergent replay

When Respec determines that the recorded and replayed process have diverged, it rolls back execution to the last checkpoint in which the recorded and replayed process states matched. A rollback *must* be performed when the replayed process tries to perform an external output that differs from the output produced by the recorded process; e.g., if the arguments to a `write` system call differ. Until such a mismatch occurs, we need not perform a rollback. However, for performance reasons, Respec also eagerly rolls back the processes when it detects a mismatch in state that makes it *unlikely* that two processes will produce equivalent external output. In particular, Respec verifies that the replayed thread makes system calls and synchronization operations with the same arguments as the recorded thread. If either the call type or arguments do not match, the processes are rolled back. In addition, at the end of each epoch, Respec compares the address space and registers of the recorded and replayed processes. Respec rolls the processes back if they differ in any value.

Checking memory values at each epoch has an additional benefit: it allows Respec to release external output for the prior epoch. By checking that the state of the recorded and the replayed process are identical, Respec ensures that it is possible for them to produce identical output in the future. Thus, Respec can commit any prior checkpoints, retaining only the one for which it just compared process state. All external output buffered prior to the retained checkpoint is released at this time. In contrast, if Respec did not compare process state before discarding prior checkpoints, it would be possible for the recorded and replayed process to have diverged in such a way that they could no longer produce the same external output. For example, they might contain different strings in an I/O buffer. The next system call, which outputs that buffer, would always externalize different strings for the two processes.

Respec leverages kernel copy-on-write mechanisms to reduce the amount of work needed to compare memory states. Since the checkpoint is an (as-yet-unexecuted) copy of the recorded process, any modifications made to pages captured by the checkpoint induce a copy-on-write page fault, during which Respec records the address of the faulted page. Similarly, if a page fault is made to a newly mapped page not captured by the checkpoint, Respec also records the faulting page. At the end of each epoch, Respec has a list of all pages modified by the recorded process. It uses an identical method to capture the pages modified by a replayed process; instead of creating a full checkpoint, however, it simply makes a copy of its address space structures to induce copy-on-write faults. Additionally, Respec parallelizes the memory comparison to reduce its latency.

In comparing address spaces, Respec must exclude the memory modified by the replay mechanism itself. It does this by placing all replay data structures in a special region of memory that is ignored during comparisons. In addition, it allocates execution stacks for user-level replay code within this region. Before entering a record/replay routine, Respec switches stacks so that stack modifications are within the ignored region. Finally, the shared user-level log, which resides in a memory region shared between the recorded and replayed process, is also ignored during comparisons.

4.5 Rollback

Rollback is triggered when memory states differ at the end of an epoch or when a mismatch in the order or arguments of system calls or synchronization operations occurs. Such mismatches are always detected by the replayed process, since it executes behind the recorded process. Respec uses Speculator to roll back the recorded process to the last checkpoint at which program states matched. Speculator switches the process, thread, and other identifiers of the process being rolled back with that of the checkpoint, allowing the checkpoint to assume the identity of the process being rolled back. It then induces the threads of the recorded process to exit. After the rollback completes, the replayed process also exits.

Immediately after a checkpoint is restored, the recorded thread creates a new replayed process. It also creates a new checkpoint using Speculator (since the old one was consumed during the rollback). Both the recorded and replayed threads then resume execution.

Given an application that contains many data races, one can imagine a scenario in which it is extremely unlikely for two executions to produce the same output. In such a scenario, Respec might enter a pathological state in which the recorded and replayed processes are continuously rolled back to the same checkpoint. We avoid this behavior by implementing a mechanism that guarantees forward progress even in the presence of unbounded data races. This mechanism is triggered when we roll back to the same checkpoint twice.

During retry, one could use a logger that instruments all memory accesses and records a precise memory order. Instead we implemented a simpler scheme. We observe that the recorded and replayed process will produce identical results for even a racy application as long as a single thread is executed at a time and thread preemptions occur at the same points in thread execution. Therefore, Respec picks only one recorded thread to execute; this thread runs until either it performs an operation that would block (e.g., a `futex` wait system call) or it executes for the epoch interval. Then, Respec takes a new checkpoint (the other recorded threads are guaranteed to be in a safe place in their execution since they have not executed since the restoration of the prior checkpoint). After the checkpoint is taken, all recorded threads continue execution. If Respec later rolls back to this new checkpoint, it selects a new thread to execute, and so on. Respec could also set a timer to interrupt user-level processes stuck in a spin loop and use a branch or instruction counter to interrupt the replayed process at an identical point in its execution; such mechanisms are commonly used in uniprocessor replay systems [10]. Thus, Respec can guarantee forward progress, but in the worst case, it can perform no better than a uniprocessor replay system. Fortunately, we have not yet seen a pathological application that triggers this mechanism frequently.

4.6 Offline replay support

When requested, Respec can optionally save information to enable an offline replay of the recorded process. This information includes the kernel's log of system calls, the user-level log of synchronization operations, and an MD5 checksum of address space and register state at the end of each committed rollback. Since not all races are logged, offline replay of the recorded process is not guaranteed

to succeed in the first attempt. However, since the recorded process has been replayed successfully at least once, it is likely that offline replay will eventually succeed, although it may require a number of rollbacks and retries. Combining Respec output with an offline replay search tool, such as is done in ODR [1] and PRES [32], would be a promising approach to reduce search time.

4.7 Security considerations

One use of deterministic replay is to parallelize security checks by running them on one or more replayed processes [28]. This section describes the security issues that must be considered when using replay in this type of adversarial context, in which the software being replayed may actively try to disrupt the replay system. For example, an attacker who compromises the replay system could try to force the replayed process to skip over the execution interval that a security check would have detected as suspicious.

Recall that the goals of our externally deterministic replay system are to ensure that: (1) the replayed execution matches the output of the original execution (including the state at the end of a replay epoch), and (2) the replayed execution is a natural execution of the target program. By a "natural execution", we mean that the replayed execution must match the instruction-for-instruction execution of one of the possible executions of the original system (i.e., the system without the kernel and library support for replay). While the replayed execution may diverge from the original execution within an epoch, it must still converge back with the state of the original execution by the end of the epoch.

Meeting these goals ensures that if the replayed process passes all security checks, a natural run exists that passes all security checks, so the output and state produced by the original and replayed processes is safe with respect to those checks. The most an attacker can do is to choose *which* natural run the replayed process executes, but that natural run must still match the output and state of the original process. For example, an attacker could try to avoid detection by a buffer overflow security check by (1) overflowing a buffer in the original process and (2) causing the replayed process to execute a different natural path that did not overflow the buffer (and hence did not trigger the security check). However, the same output and state could have been produced by the program *without* the buffer overflow, since that is exactly what the replayed process has done.

We next argue that Respec meets these goals, even when replaying malicious software. We assume that the software being recorded and replayed cannot corrupt kernel data. Therefore, we place as much of Respec as possible within the kernel. Speculator, the kernel log, and memory checkpoints are all placed in the kernel.

The only replay data that the malicious software can corrupt is the user-level log shared between the recorded and replayed processes; this log contains the order of user-level synchronization operations. Respec must therefore treat the user-level log as suspect. If, for example, Respec trusted the header length fields in each user-level log record, the malicious software might be able to cause a buffer overflow in the replay system and cause the replayed process to deviate from the set of natural runs. More subtly, the malicious software could record an order of synchronization operations that could not be generated by a natural run. For example, the malicious software could write a log record that caused a lock operation to succeed even when the lock was already held by another thread.

To protect against these attacks, Respec has an optional verification mode that can be used during replay. When verification mode is enabled, the replayed process copies each record from the shared user-level log to a non-shared memory region, then verifies that the log record represents a *possible* execution path. For instance, it rejects a log record that shows a lock operation completing successfully when the lock is already held by another thread. To verify records, the replayed process shadows the state of locks, futexes, and other logged data structures (the original memory reserved for these structures is used for this purpose since the replayed process does not execute the actual synchronization operations). If a logged action would be invalid given the shadowed state, the action is rejected and a mismatch reported. Since Respec only logs a few types of low-level operations, such verification is not difficult. However, since verification adds overhead, it can be disabled to improve performance when replay is not being used for security checks.

5. Evaluation

Our evaluation answers the following questions:

- What is the overhead of Respec record and replay for common applications and benchmarks?
- How often does imprecise logging lead to rollbacks for those application and benchmarks?
- What is the cost of rollback and retry when it occurs?

5.1 Methodology

We ran all experiments on a 2 GHz 8-core Xeon processor with 3 GB of RAM running CentOS Linux version 5.3. The Linux kernel is a stock Linux 2.6.27 kernel, which we modified to support Respec deterministic replay. In addition, the kernel has the Speculator support for speculative execution. We also modified the GNU glibc library version 2.5.1 to support Respec.

We used three sets of benchmarks. The first set is five benchmarks from the PARSEC suite [4]: blackscholes, bodytrack, fluidanimate, swaptions, and streamcluster. The second set is six benchmarks from the SPLASH-2 suite [39]: ocean, raytrace, volrend, water-nsq, fft, and radix. The final set is four parallel applications: pbzip2, which we use to compress a 17 MB log file in parallel; pfscan, which we use to search in parallel for a string in a directory with 952 MB of log files; aget, which we use to retrieve a 21 MB file over a local network; and Apache, which we test using ab (Apache Bench) to simultaneously send 100 requests each from four concurrent clients over a local network.

For all benchmarks, we ensure that all files are in the file cache in kernel memory before execution begins. Thus, our experimental results do not include any disk I/O time, which would mask the relative overhead of deterministic replay. We report three values for each experiment: the original execution time of the application running on a stock system, a "redundant" execution time in which two copies of the application are run concurrently on a stock system by forking an execution at the start, and the execution time using Respec to provide deterministic replay. The redundant execution is a lower bound on execution time for online replay; of course, the execution outputs could diverge. The Respec execution time measures the time for both the recorded and replayed processes to finish.

For each benchmark, we vary the number of worker threads from one to four. Many benchmarks have additional control threads which do little work during the execution; we do not count these in the number of threads. Pbzip2 uses two additional threads: one to read file data and one to write the output; these threads are also not counted in the number of threads shown. Unless otherwise mentioned, all results are the mean of ten trials.

5.2 Record and replay performance

Table 1 shows the overall performance results for Respec. The first two columns show the application or benchmark executed and the

application	threads	synch. ops.	system calls	epochs	pages compared	original time (s) & stdev.	redundant time (s) & stdev.	Respec time (s) & stdev.	slowdown wrt redundant	slowdown wrt original
blackscholes	1	524330	30	4	2973	7.04 (0.00)	7.04 (0.00)	7.34 (0.02)	4%	4%
	2	524345	36	4	2974	3.54 (0.00)	3.54 (0.00)	3.79 (0.04)	7%	7%
	3	524361	41	4	2976	2.37 (0.00)	2.37 (0.00)	2.77 (0.05)	17%	17%
	4	524376	47	4	2977	1.79 (0.00)	1.94 (0.08)	2.24 (0.08)	15%	25%
bodytrack	1	387794	6180	11	4091	8.60 (0.01)	8.58 (0.03)	8.74 (0.02)	2%	2%
	2	389907	7539	11	4235	4.89 (0.01)	4.83 (0.08)	5.13 (0.06)	6%	5%
	3	393314	9982	7	3469	3.55 (0.02)	3.53 (0.05)	3.82 (0.13)	8%	8%
	4	395835	11060	10	5733	2.86 (0.02)	3.08 (0.09)	4.82 (0.21)	56%	68%
fluidanimate	1	541964	2749	5	26475	6.69 (0.00)	6.72 (0.04)	6.74 (0.01)	0%	1%
	2	4773663	3017	5	28309	4.04 (0.00)	4.13 (0.02)	4.60 (0.03)	11%	14%
	4	7545571	3136	5	28895	2.52 (0.01)	2.61 (0.01)	4.35 (0.19)	67%	73%
swaptions	1	1922355	102	7	413	6.77 (0.00)	6.78 (0.00)	7.87 (0.04)	16%	16%
	2	1905088	214	6	447	3.39 (0.00)	3.40 (0.01)	3.89 (0.06)	15%	15%
	3	1905178	286	4	464	2.34 (0.00)	2.34 (0.01)	2.56 (0.03)	10%	10%
	4	1800897	818	4	471	1.76 (0.18)	1.78 (0.02)	2.06 (0.16)	16%	17%
streamcluster	1	59681	7239	14	3121	10.97 (0.08)	11.09 (1.41)	11.01 (0.66)	-1%	0%
	2	108360	31159	8	2769	5.62 (0.66)	5.66 (0.47)	5.62 (0.56)	-1%	0%
	3	157874	56674	8	2798	3.35 (0.53)	4.64 (0.54)	5.68 (0.56)	22%	69%
	4	208367	83029	8	2834	2.51 (0.32)	4.63 (0.37)	5.88 (0.52)	27%	134%
ocean	1	2648	61	5	202790	4.93 (0.00)	5.00 (0.00)	7.06 (0.05)	41%	43%
	2	5203	855	4	154503	2.50 (0.00)	3.03 (0.01)	4.29 (0.03)	42%	72%
	4	10366	2642	3	106102	1.49 (0.03)	2.28 (0.02)	3.25 (0.08)	43%	119%
raytrace	1	1115639	1143	2	8352	0.79 (0.01)	0.80 (0.01)	1.34 (0.01)	68%	70%
	2	1123858	5632	2	8352	0.64 (0.01)	0.65 (0.01)	1.29 (0.03)	99%	101%
	3	1117220	7046	2	8362	0.59 (0.00)	0.59 (0.00)	1.47 (0.12)	148%	150%
	4	1118038	6102	2	8352	0.57 (0.00)	0.57 (0.00)	1.55 (0.10)	171%	173%
volrend	1	632	448	2	7372	1.83 (0.01)	1.83 (0.01)	1.88 (0.01)	3%	3%
	2	141678	619	2	7377	1.33 (0.01)	1.34 (0.00)	1.39 (0.01)	3%	4%
	3	143319	2784	2	7381	1.19 (0.00)	1.19 (0.00)	1.27 (0.00)	7%	7%
	4	142177	4084	2	7385	1.10 (0.00)	1.13 (0.06)	1.21 (0.05)	7%	9%
water-nsq	1	67055	535	3	4760	3.05 (0.11)	3.09 (0.10)	3.16 (0.13)	2%	4%
	2	122848	908	2	4203	1.68 (0.03)	1.68 (0.03)	1.74 (0.02)	4%	4%
	3	156131	1281	2	6254	1.14 (0.01)	1.23 (0.03)	1.34 (0.03)	9%	18%
	4	181343	1840	2	8305	0.90 (0.02)	0.91 (0.01)	1.38 (0.01)	52%	54%
fft	1	110	22	1	0	0.82 (0.00)	0.84 (0.09)	0.84 (0.00)	1%	2%
	2	166	41	1	0	0.53 (0.00)	0.56 (0.04)	0.57 (0.00)	1%	8%
	4	275	77	1	0	0.40 (0.01)	0.44 (0.01)	0.45 (0.00)	3%	12%
radix	1	109	18	2	16408	4.50 (0.01)	4.51 (0.65)	4.61 (0.02)	2%	3%
	2	193	35	2	16416	2.29 (0.01)	2.35 (0.01)	2.41 (0.03)	3%	5%
	4	392	81	2	31206	1.16 (0.00)	1.28 (0.00)	1.44 (0.04)	12%	24%
pfscan	1	267	75	3	19	1.94 (0.01)	1.94 (0.01)	1.99 (0.05)	3%	2%
	2	301	83	2	15	1.15 (0.01)	1.16 (0.03)	1.19 (0.03)	3%	4%
	3	340	91	2	16	0.92 (0.00)	0.94 (0.01)	0.97 (0.02)	3%	5%
	4	376	99	2	16	0.77 (0.00)	0.83 (0.02)	0.98 (0.04)	19%	28%
pbzip2	1	993	297	20	33654	4.59 (0.00)	4.81 (0.16)	4.83 (0.10)	0%	5%
	2	958	359	11	33699	2.35 (0.00)	2.42 (0.09)	2.73 (0.13)	13%	16%
	3	954	386	8	33794	1.64 (0.05)	1.70 (0.03)	2.03 (0.15)	19%	24%
	4	1005	409	6	33050	1.33 (0.00)	1.44 (0.04)	1.93 (0.23)	34%	45%
aget	1	8618	14681	4147	29039	2.05 (0.16)	N/A	2.19 (0.14)	N/A	7%
	2	8739	13905	3921	27985	1.93 (0.00)	N/A	2.17 (0.08)	N/A	13%
	3	8770	13096	3689	26348	1.94 (0.00)	N/A	2.08 (0.04)	N/A	7%
	4	8432	12944	3642	26269	1.96 (0.04)	N/A	2.08 (0.06)	N/A	6%
apache	1	3808	11114	18065	5654	8.08 (0.06)	N/A	8.13 (0.02)	N/A	1%
	2	3557	10919	17898	5851	7.89 (0.03)	N/A	8.56 (0.12)	N/A	9%
	3	3417	10902	17922	5899	7.40 (0.04)	N/A	9.42 (0.16)	N/A	27%
	4	3571	10945	17937	5824	6.98 (0.04)	N/A	10.04 (0.12)	N/A	44%

Table 1. Respec performance. Results are the mean of ten trials with the exception of pbzip2 and aget, which show the mean of 100 and 50 trials respectively. Values in parentheses show standard deviations.

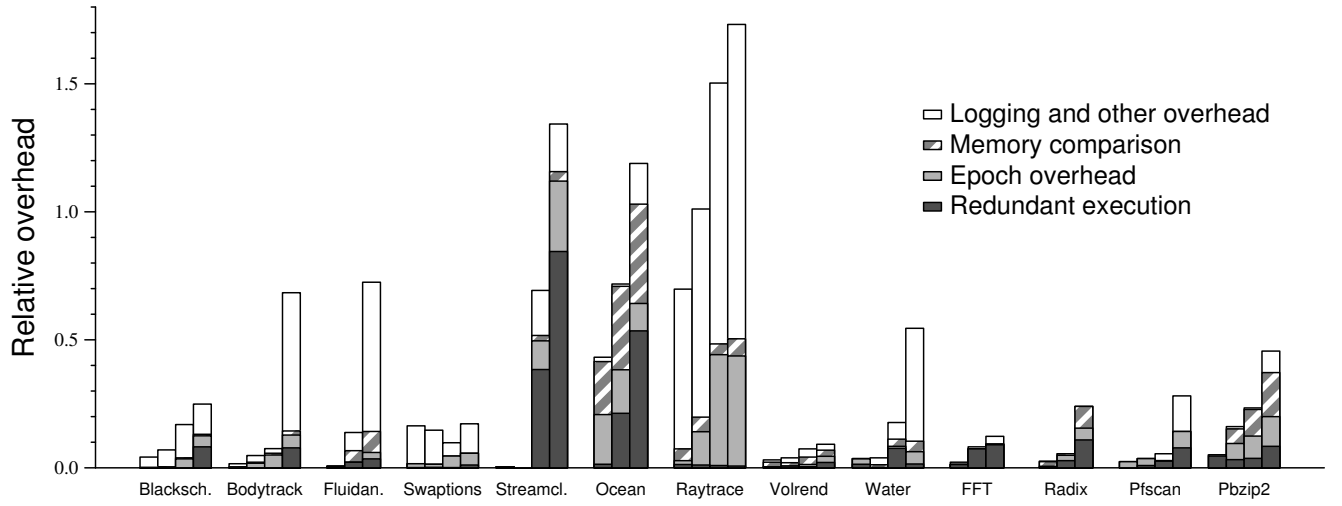

Figure 4. Breakdown of overhead per benchmark. For most benchmarks, results are presented for 1, 2, 3, and 4 threads (left to right). For Fluidanimate, Ocean, FFT, and Radix, results are presented for 1, 2, and 4 threads.

threads	rollback freq	original time (s) & stdev.	type	Respec time (s) & stdev.	slowdown wrt original
1	0%	4.59 (0.00)	overall	4.83 (0.10)	5%
2	13% once	2.35 (0.00)	w/o rollback	2.70 (0.09)	15%
			w/ rollback	2.97 (0.12)	26%
			overall	2.73 (0.13)	16%
3	9% once 2% twice	1.64 (0.05)	w/o rollback	2.00 (0.10)	22%
			w/ rollback	2.29 (0.17)	40%
			overall	2.03 (0.15)	24%
4	15% once 1% twice	1.33 (0.00)	w/o rollback	1.88 (0.16)	41%
			w/ rollback	2.24 (0.29)	68%
			overall	1.93 (0.23)	45%

Table 2. Rollback frequency in pbzip2

threads	rollback freq	original time (s) & stdev.	type	Respec time (s) & stdev.	slowdown wrt original
1	10% once 2% twice	2.05 (0.16)	w/o rollback	2.19 (0.14)	7%
			w/ rollback	2.21 (0.13)	8%
			overall	2.19 (0.14)	7%
2	20% once 2% twice	1.93 (0.00)	w/o rollback	2.17 (0.08)	13%
			w/ rollback	2.17 (0.05)	13%
			overall	2.17 (0.08)	13%
3	24% once	1.94 (0.00)	w/o rollback	2.08 (0.05)	7%
			w/ rollback	2.09 (0.02)	8%
			overall	2.08 (0.04)	7%
4	18% once 2% twice	1.96 (0.04)	w/o rollback	2.07 (0.05)	6%
			w/ rollback	2.08 (0.02)	6%
			overall	2.08 (0.06)	6%

Table 3. Rollback frequency in aget

number of worker threads used. The next four columns give statistics about Respec execution: the number of user-level synchronization operations logged, system calls logged, epochs executed, and memory pages compared. The next three columns show the original, redundant, and Respec execution times. The last two columns show Respec's overhead with respect to the lower bound of the redundant execution time and with respect to the original execution time. For the two networked applications (aget and apache), measuring redundant execution time is difficult because the two separate processes contend for network resources, whereas with Respec only the recorded process sends and receives network packets.

Figure 4 provides a breakdown of the overhead normalized to the original execution time for all non-networked benchmarks. Each data set shows results for 1, 2, 3 (if feasible), and 4 worker threads. The dark shaded area in each bar show relative overheads for redundant execution. The lighter shaded areas show relative overhead associated with use of multiple epochs, excluding memory comparison cost. We believe that this overhead is mainly due to page fault overhead. The diagonally hashed areas show relative overhead due to memory comparison. The remaining region shows all other overhead, the majority of which is likely due to logging synchronization operation and system calls. Respec's implementation makes it difficult for us to provide similar breakdowns for networked applications (Apache and aget) because use of multiple epochs is required to correctly interact with clients on different computers, due to the output commit problem (as discussed in Section 4.2).

Examining the results, we see that Respec overhead is generally quite low. For 2 worker threads, Respec has average overhead with respect to the original execution of only 18% across all benchmarks. This overhead gradually increases with the number of threads; with 4 threads, Respec's average overhead is 55%. Compared to the lower bound of redundant execution, Respec's average overhead is 16% with 2 threads and 40% with 4 threads. We conjecture that this increase derives mostly from the increased synchronization between replay threads.

It is informative to examine some of the benchmarks with extreme characteristics. Fluidanimate from the PARSEC suite and raytrace from the SPLASH-2 suite execute over two million logged synchronizations per second with four threads. Most of these operations are uncontended lock and unlock operations, which do not require a system call. In these cases, Respec overhead derives from the cost of logging these user-level operations. Ocean and streamcluster show larger overheads with respect to the original execution for 4 threads, but show significantly less overhead for 2 threads. In fact, for 4 threads, simply executing two copies of these benchmarks concurrently shows similar large increases in execution time, indicating that most of the overhead derives simply from sharing

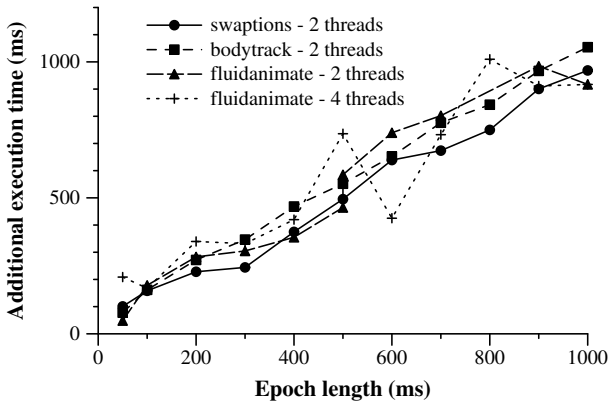

Figure 5. Impact of epoch interval on rollback overhead

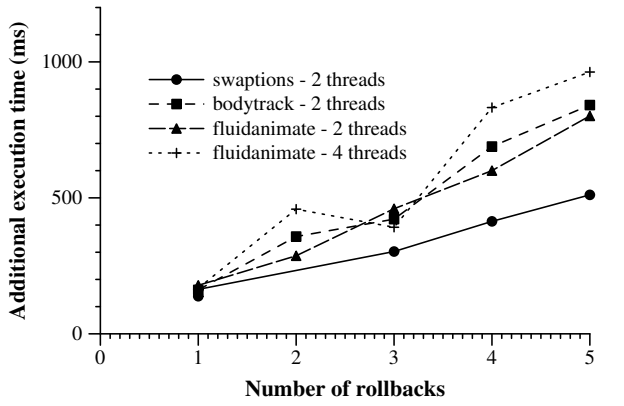

Figure 6. Impact of number of rollbacks on rollback overhead

the limited memory bandwidth and processor caches for redundant execution rather than from Respec itself. Many benchmarks show a spike in overhead when the number of worker threads increases from 3 to 4; part of the reason for this spike may be that our 8 core machine has no CPU capacity to spare for auxiliary threads if 4 cores each are used to record and replay worker threads.

Respec overhead for the four applications at the bottom of Table 1 is relatively low, averaging 11% with 2 worker threads and 31% with 4 worker threads. This lower overhead is to be expected since these applications issue fewer system calls and synchronization operations than the PARSEC and SPLASH-2 benchmarks.

5.3 Rollback frequency

Rollbacks were infrequent events for our benchmarks. In fact, only pbzip2 and aget were rolled back during our experiments. Pbzip2 has one benign application-level data race in which an output thread repeatedly spins on two variables (once for each chunk of data being compressed) waiting for a worker thread to modify them from zero to one. While the race is benign, it affects the number of system calls issued in some executions. It would also be difficult to identify and log this race other than through manual code inspection since the spin loop does not use atomic instructions. Additionally, pbzip2 uses the stdlibc++ library, which we have not modified to log synchronization operations.

To better understand the frequency of rollbacks, we ran pbzip2 100 times. Table 2 shows that 13–16% of the executions with more than one worker thread contained one rollback. In general, the cost of rollback was reasonable. Rollbacks contribute 8% of Respec's total overhead when pbzip2 uses multiple worker threads.

For aget, one thread reads and displays download progress without obtaining a lock. This data race is benign since display of slightly stale status information is acceptable. Table 3 shows that the divergent output and memory state leads to rollbacks. However, the performance impact of these rollbacks is negligible because Respec checkpoints aget very frequently (on every network receive).

5.4 The cost of rollback

To better understand the cost of rollbacks, we *artificially* inserted rollbacks into some of our benchmarks by emulating the failure of a divergence check during benchmark execution. We disabled Respec's adaptive epoch algorithm and manually set the epoch interval to a configured value.

Figure 5 shows the additional time needed to complete four benchmarks when a single rollback is artificially introduced for different epoch intervals. The results show that rollback overhead is roughly proportional to the length of the epoch interval. Intuitively, the execution time increases by the amount of work that must be redone after a rollback. Any fixed cost introduced by the rollback mechanism itself appears to be minimal.

For a final experiment, we varied the number of rollbacks during benchmark execution while keeping the epoch interval fixed at 100 ms. Figure 6 shows that the cost of rollbacks is proportional to the number of rollbacks. Interestingly, fluidanimate and bodytrack show a cost of approximately 160–180 ms per rollback (60–80 ms greater than the epoch interval length). Upon further investigation, we found the reason to be our barrier implementation for checkpointing; it can take several tens of milliseconds for all threads to reach the barrier for these two benchmarks.

6. Related work

To the best of our knowledge, Respec is the first system to support low-overhead, online deterministic replay of multithreaded programs on commodity shared memory multiprocessors without hardware support.

There have been several systems developed over the last two decades to record and replay a program's execution, primarily for debugging. IGOR [12], one of the earliest recorders, uses copy-on-write checkpointing support in the operating system to record and reproduce an intermediate state of a process. In addition to checkpointing support, to ensure deterministic replay of a program from a particular state, it is also necessary to record non-deterministic system events such as interrupts, DMA, and also the values of any non-deterministic instructions such as the x86 RDTSC (ReaD TimeStamp Counter) instruction. These events can be recorded in any of the layers in the software system stack. Systems like Hyperviser [7], Boothe [6], and Flashback [36] instrument the operating system to record and replay the non-deterministic events. DejaVu [8] and jRapture [37] record most (but not all) of the non-deterministic system events by instrumenting the Java Virtual Machine (JVM). ReVirt [10] and ReTrace [42] use support in the virtual machine monitor that interfaces between the guest and host operating systems. Unlike Respec, none of these systems support multiprocessor replay, because they cannot record and replay the non-deterministic order between shared memory accesses executed by concurrent threads.

Systems like ReVirt [10] and DejaVu [8] support replay of multithreaded programs on a uniprocessor system by deterministically replaying the thread schedules. But, replay of a multithreaded pro-

gram on a multiprocessor has remained a difficult problem. One of the first systems to address this problem is InstantReplay [17]. It instruments every memory access to a shared object to record the order in which different threads accessed it. Recent systems such as PinSel [24] and Microsoft's iDNA [3] also instrument every memory access to enable multiprocessor replay. But, monitoring every memory access is expensive (iDNA [3], for instance, is about 5–15x slower than the native execution). Instead of monitoring every memory access, SMP-ReVirt [11] uses memory protection bits to detect all the shared memory dependencies and recorded the memory order. But, handling a memory protection fault for every shared memory dependency is also inefficient (up to 9x slower).

Instead of recording the order of all shared memory accesses, RecPlay [34] and JaRec [14] instrument just the synchronization operations and recorded their order of execution. This approach only ensures deterministic replay of a program up until the first data race, which limits the use of a replayer in many ways. For example, while debugging using a replayer, a programmer might want to understand the after effects of a data race bug in order to triage it, which is not possible with RecPlay. After finding a data race bug, a tester might not want to wait for the developer to fix it before he/she could carry on with further tests. Also, for continuously checking the correctness of production runs it is necessary to replay past the first data race. In fact, most real world applications contain benign data races [25]. For such applications, a replay tool is most useful only if it can replay past the benign data races.

Any software-only solution for recording a precise order of all shared memory accesses is likely to be expensive. To reduce the performance cost, processor support could be used. Bacon and Goldstein [2] observed that the memory order can be determined from the coherence messages, and so they proposed a hardware design to log all the coherence messages. Netzer proposed a transitive reduction algorithm [26] to reduce the number of memory order logs that need to be recorded. Recently, there has been significant advancements in processor-based deterministic replay support [15, 21–23, 40, 41]. They are all based on Bacon and Goldstein's approach [2], requiring invasive changes to the coherence mechanism. Unfortunately, the parts that ensure coherence are some of the hard-to-verify components in a multicore processor [35]. Lee et al. [18] recently proposed a multiprocessor replay solution based on offline symbolic analysis that does not require a precise shared memory dependency recorder, but it still requires hardware support. While a hardware solution could be very efficient, it would require intense lobbying from the software developers and several years before hardware vendors decide to include such a specialized feature.

ODR [1] and PRES [32] are most similar to Respec in that they also log less information than is necessary to guarantee deterministic replay of a multithreaded program. Because these systems focus on offline replay, they are able to conduct an expensive search (during replay) through the possible interleavings of shared memory accesses to find an order that produces the recorded output. Online replay requires a faster replay mechanism, which we achieve through speculative execution of the recorded process. In addition, the external deterministic replay provided by Respec ensures that the complete program state is identical, whereas ODR ensures only that the output is identical. Respec's more stringent definition of equivalence is needed to commit checkpoints and release output, since otherwise an undetected memory difference could later lead deterministically to divergent output. Finally, ODR's or PRES's search algorithm could use data logged by Respec, such as system calls, synchronization operations, and a hash of the address space and registers at each epoch. The address space hashes in particular would provide fixed states that could help guide offline search.

Speck [28] and Respec both use speculative execution during replay, but for different purposes. Respec uses speculation to support multiprocessor replay, a feature not supported at all by Speck. Speck uses speculation to parallelize robustness checks. There is potential synergy between the two approaches, and, in fact, Speck is one of the potential use cases we posit for online replay.

7. Conclusion

With the advent of multicore processors, introspective tools that help us understand and verify parallel programs are needed. A deterministic record and replay system could serve as a foundation for building many such tools by overcoming the inherent non-determinism in a multiprocessor system. Unfortunately, an efficient software solution for multiprocessor replay has proven elusive.

Respec addresses this need by providing a solution for fast, online shared memory multiprocessor replay. Respec uses two novel techniques to achieve efficiency: external determinism, a new fidelity level for replay, and speculative execution. External determinism provides adequate guarantees for most applications of replay, but its relaxed constraints yield sufficient freedom to support efficient multiprocessor replay. Respec uses speculative execution to optimistically log only the most common synchronization operations, relying on rollback and retry to guarantee correctness in the rare cases where the recorded and replayed processes diverged due to unlogged races. These two techniques allow Respec to concurrently record and replay multithreaded programs with an average overhead of 18% for two threads and 55% for four threads.

Acknowledgments

We thank the anonymous reviewers for comments that improved this paper. The work is supported by the National Science Foundation under award CNS-0905149. Peter Chen is supported by NSF award CNS-0614985 and Intel Corporation. Jason Flinn is supported by NSF CAREER award CNS-0346686. Satish Narayanasamy is supported by NSF award CCF-0916770 and Microsoft. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, the U.S. government, or industrial sponsors.

References

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[2] D. F. Bacon and S. C. Goldstein. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206. ACM Press, 1991.

[3] S. Bhansali, W. Chen, S. de Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*, June 2006.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[5] H. J. Boehm and S. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of PLDI*, pages 68–78. ACM, 2008.

[6] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, pages 299–310, 2000.

[7] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[8] J. D. Choi, B. Alpern, T. Ngo, and M. Sridharan. A perturbation free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[9] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Technical Conference*, pages 1–14, June 2008.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[11] G. W. Dunlap, D. G. Lucchetti, M. Fetterman, and P. M. Chen. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 121–130, March 2008.

[12] S. I. Feldman and C. B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, 1988.

[13] K. Fraser and F. Chang. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference*, pages 325–338, San Antonio, TX, June 2003.

[14] A. Georges, M. Christiaens, M. Ronsse, and K. D. Bosschere. Jarec: A portable record/replay environment for multi-threaded java applications. In *Software: Practice and Experience*, 2004.

[15] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight memory Race Recording. In *Proceedings of the 2008 International Symposium on Computer Architecture*, pages 265–276, June 2008.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 1–15, April 2005.

[17] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, 36(4):471–482, 1987.

[18] D. Lee, M. Said, S. Narayanasamy, Z. J. Yang, and C. Pereira. Offline Symbolic Analysis for Multi-Processor Execution Replay. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[19] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[20] J. Manson, W. Pugh, and S. Adve. The java memory model. In *Proceedings of POPL*, pages 378–391. ACM, 2005.

[21] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently . In *Proceedings of the 2008 International Symposium on Computer Architecture*, pages 289–300, June 2008.

[22] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 73–84, 2009.

[23] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 229–240, 2006.

[24] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, June 2006.

[25] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, June 2007.

[26] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[27] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 191–205, Brighton, United Kingdom, October 2005.

[28] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[29] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 1–14, Seattle, WA, October 2006.

[30] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

[31] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 361–376, Boston, MA, December 2002.

[32] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. Do you have to reproduce the bug at the first replay attempt? – PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[33] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 235–248, Brighton, United Kingdom, October 2005.

[34] M. Ronsse and K. D. Bosschere. RecPlay: A Full Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[35] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro Top Picks*, 27(1):12–25, 2007.

[36] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, June 2004.

[37] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[38] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of the 2007 Symposium on Operating Systems Principles*, pages 131–144, October 2007.

[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[40] M. Xu, R. Bodik, and M. D. Hill. A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture*, June 2003.

[41] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 49–60, 2006.

[42] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.