Appears in the Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30), June 2003.

ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes *

Milos Prvulovic and **Josep Torrellas**
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

While removing software bugs consumes vast amounts of human time, hardware support for debugging in modern computers remains rudimentary. Fortunately, we show that mechanisms for Thread-Level Speculation (TLS) can be reused to boost debugging productivity. Most notably, TLS's rollback capabilities can be extended to support rolling back recent buggy execution and repeating it as many times as necessary until the bug is fully characterized. These incremental re-executions are deterministic even in multithreaded codes. Importantly, this operation can be done automatically on the fly, and is compatible with production runs.

As a specific implementation of a TLS-based debugging framework, we introduce ReEnact. ReEnact targets a particularly hairy class of bugs: data races in multithreaded programs. ReEnact extends the communication monitoring mechanisms in TLS to also detect data races. It extends TLS's rollback capabilities to be able to roll back and *deterministically* re-execute the code with races to obtain the race signature. Finally, the signature is compared to a library of race patterns and, if a match occurs, the execution may be repaired. Overall, ReEnact successfully detects, characterizes, and often repairs races automatically on the fly. Moreover, it is fully compatible with always-on use in *production runs*: the slowdown of race-free execution with ReEnact is on average only 5.8%.

1. Introduction

As the computing industry continues to mature, the issues of reliability, maintainability, cost of ownership, and end-user satisfaction are receiving more attention. As a result, there is now more emphasis on devising novel architectural support for avoiding or tolerating hardware failures (e.g. [2, 16, 20, 24]). While this is a welcome trend, we note that some studies show that software failures account for as much as 40% of computer system failures [14].

Unfortunately, removing software bugs is a task that requires enormous human labor. Entire teams of people are dedicated to test the software and look for anomalies. These anomalies are reported to developers who attempt to find the bugs that cause them. Despite this vast effort, software released to end users still contains numerous bugs. These bugs continue to consume human time in the form of software problems at the user site, user-vendor communication, and subsequent "bug-fix" software releases.

*This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM and Intel. Milos Prvulovic is supported by an Intel PhD Fellowship.

Surprisingly enough, despite all of this, the hardware support for debugging in modern computers remains rudimentary. Typically, it is limited to support for interrupting the program if it accesses data from a specified address, or if it executes a specified instruction [12]. Many software tools to aid debugging have been developed (e.g. GDB [25]), which make the best use of the limited hardware support. Unfortunately, most sophisticated tools have to rely on software instrumentation to perform their checks, which degrades performance to a level that makes them incompatible with production runs. If we could design effective debugging hardware that remained on while in production runs, human time to debug programs would be reduced drastically.

Fortunately, well-known architectural support for Thread-Level Speculation (TLS) [5, 10, 11, 19, 26, 27] can be reused for software debugging. Previous work has pointed out that TLS's ability to squash or commit the side effects of code sections as a unit can be used to enhance software reliability [17]. In this paper, we claim that TLS has the potential to be a core technology for software debugging. Specifically, with simple extensions, it supports the ability to roll back recent buggy execution and repeat it as many times as necessary until the bug is fully characterized. These incremental re-executions are deterministic even in multithreaded codes. Importantly, this operation can be done automatically on the fly, and is compatible with production runs.

One particularly hairy class of bugs is data races in multithreaded programs. Data races occur when threads access shared data in an unsynchronized manner. In this case, program execution has some unintended non-determinism that is typically considered a bug. Race bugs are notoriously difficult to reproduce and debug. Consequently, we use them as the problem domain to flesh out our ideas.

In this paper, we introduce ReEnact, a specific implementation of a TLS-based debug framework that targets data races. ReEnact detects, characterizes in detail, and often repairs races automatically on the fly. It extends the communication monitoring mechanisms in TLS to also detect data races. It enhances TLS's rollback capabilities to be able to roll back and *deterministically* re-execute the code with races to obtain the race signature. Finally, the signature is compared to a library of race patterns and, if a match occurs, the execution may be repaired. Our experiments using SPLASH-2 applications show that ReEnact is very effective: it successfully detects and characterizes data races, many of which are also repaired on the fly. Moreover, ReEnact is fully compatible with always-on use in *production runs*: the slowdown of race-free execution with ReEnact is on average only 5.8%.

This paper is organized as follows: Section 2 describes the challenges in debugging software; Section 3 describes how TLS

mechanisms can be reused for debugging; Section 4 describes ReEnact; Section 5 addresses some implementation issues in ReEnact; Section 6 describes how we evaluate ReEnact; Section 7 presents the results of the evaluation; Section 8 discusses related work and, finally, Section 9 concludes.

2. Challenges in Debugging Software

Removing a software bug typically involves three phases: *detection*, *characterization*, and *repair*. A bug is *detected* by observing one of its symptoms, such as program termination with a fault or incorrect results. Bug *characterization* determines the cause of the undesired behavior. Characterization typically involves backtracking from the point of detection to the actual cause of the problem, and is usually done by placing breakpoints in "strategic" places in the code, re-running the program, and examining the state in these places. When the cause of the problem has been determined, the program is *repaired* and re-executed.

All three phases of debugging often require extensive human involvement. Indeed, detection often requires careful cross-examination of program inputs and results. Characterization often involves many time-consuming iterations of the instrument-and-rerun process. Finally, repair may require significant modifications of the code.

To make matters worse, different phases of debugging are often performed in different environments. For example, bugs are typically detected by testers and end users, but characterized and repaired by programmers. As a result, it may be difficult for programmers to reconstruct the circumstances under which the bug was detected. Moreover, the program inputs that resulted in bug detection may contain sensitive information that the user is unwilling to disclose to the programmer. Furthermore, in multithreaded programs, some bugs such as data races are difficult to repeat even when the inputs are known and the characterization is done in the same environment. This is because the bug may depend on the particular interleaving of actions from different threads in a way that is outside direct program control.

To make debugging easier, we propose the following approach. First, since bugs may be difficult to repeat, debugging should be done *on the fly*, characterizing and possibly even repairing each bug when and where it is detected. Second, since bugs in multithreaded programs may depend on the particular interleaving of actions from different threads, bug characterization should be done via *deterministic* re-execution of recently-executed code. Finally, since bugs occurring in production runs can be costly and difficult to repeat, debugging should be always-on, even in *production runs*.

This approach to debugging requires certain support. First, on-the-fly bug characterization requires the ability to buffer recently-executed code and incrementally roll it back and re-execute it; if we had to re-run the entire program, the circumstances that triggered the bug could change. Second, to ensure deterministic re-execution in multithreaded environments, a mechanism is needed to monitor the ordering of actions from different threads as code executes; upon bug detection and state roll-back, the code is re-executed using the collected ordering information. Finally, always-on debugging in production runs mandates very low overheads in bug-free execution – users are unlikely to tolerate noticeable performance loss in production runs. Since software-only approaches to buffering and monitoring the execution order do not meet this requirement, we need special hardware support. In the rest of the paper, we present a design that fulfills these requirements.

3. Reusing TLS Mechanisms for Debugging

TLS is a technique that has been proposed for speculatively parallelizing sequential programs [5, 10, 11, 19, 26, 27]. However, we claim that its architectural mechanisms can be easily adapted to provide support for our debugging requirements. In this section, we first give an overview of TLS and then describe how it can be adapted to provide the desired debugging supports.

3.1. Brief Overview of TLS

With TLS, a sequential program is divided into a sequence of *epochs* (also called tasks, slices, or micro-threads), each of which contains many dynamic instructions. These epochs are then executed speculatively in parallel. The sequential semantics of the program creates dependences between epochs. If any such dependence is violated at run time by the parallel execution, TLS rolls back the incorrectly-executed epochs and re-executes them. To be able to do this, TLS saves the architectural register state at the beginning of each epoch and, as the epoch executes, keeps its memory state buffered, typically in the cache. With this support, an epoch is rolled back (squashed) by invalidating its buffered memory state from the cache and restoring its saved register state. An epoch that can still be squashed is *speculative*. Once an epoch is known to have executed correctly, it becomes *non-speculative*. Non-speculative epochs can *commit* in order by freeing their saved register state and allowing their buffered memory state to be merged with the architectural state of the system in main memory.

3.1.1. State Buffering. For each epoch that is still uncommitted, the processor contains a copy of the initial state of the architectural registers. This copy is not used in normal operation: it is generated when the epoch begins, read when the epoch is squashed, and freed when the epoch commits.

The memory state of an uncommitted epoch is typically buffered in the cache. In this paper, we assume that a given cache can contain state from multiple uncommitted epochs [8]. In this case, each cache line is tagged with the ID of the epoch to which the line belongs. To track dependences, each cache line also contains two status bits for each word: the Write and Exposed-Read bits. The former is set when the word is written by the epoch; the latter is set when the epoch reads the word without first writing to it.

When an epoch accesses a memory line for the epoch's first time, a new cache line is allocated and tagged with the epoch's ID. If the cache already contained the line from an earlier epoch, a new copy of the line is made in the cache, with an updated word if the access was a write [19]. This cache support enables multiple uncommitted epochs to buffer overlapping data. A drawback is that older line versions consume cache space, even though typically only the latest line version is useful for future accesses.

3.1.2. Squashing and Committing. When an epoch is squashed, all cache lines tagged with its ID are marked as invalid. In this paper, we do this by examining all the lines in the cache one by

one. This process can take up to a few thousand cycles, but it does not occur very often.

When an epoch commits, its buffered cache state is merged with the main memory *lazily*. Specifically, its cache lines stay in the cache until they are displaced or requested externally. Note, however, that displacements of different dirty versions of the same line from different (or the same) caches should not happen out of epoch-ID order. This is prevented with special coherence protocol support that ensures that memory is updated in order [19].

3.1.3. Dependence Tracking. When an epoch issues a read, the cache is searched for a line with matching address and epoch ID. If such a line is not found, or if the Write and Exposed-Read bits of the requested word are clear, the access is an exposed read. Consequently, the coherence protocol must find the correct version of the word and bring it to the cache. Such a version is the one generated by the closest predecessor epoch that wrote to it. For that, all sharer caches are interrogated with a request tagged with the ID of the reader epoch. The cache coherence protocol ensures that the requesting cache ends up with the closest predecessor version of the word.

When an epoch issues a write, a message tagged with the ID of the writing epoch is sent to all the sharers of the line. When the message is received by a sharer, the ID in the message is compared to that of the matching cached line(s). If the cached line is from a successor epoch and the Exposed-Read bit of the word is set, a dependence violation has occurred – rather than waiting for the predecessor write, the successor epoch has read the word prematurely and, therefore, has to be squashed.

In the TLS protocol that we use, dependences are tracked at the granularity of words thanks to the per-word Write and Exposed-Read bits. Per-word tracking prevents false sharing from causing unnecessary squashes. However, it can cause substantial traffic as individual coherence messages have to be sent for each of the words of a cache line. To minimize this problem, we use the optimization described in [19], which uses high-level access behavior to filter out unnecessary per-word coherence actions. For example, with this optimization, a cache miss can trigger the loading of an entire memory line into the cache.

3.2. Adapting TLS for Code Rollback

The mechanisms for in-cache buffering and rollback present in TLS can be reused to support our requirement for incremental rollback and re-execution of the buggy code. To illustrate how, consider for now only a single thread executing on a single processor. We divide this thread into epochs that execute sequentially. When a new epoch starts, the register state is saved. From then on, all memory accesses are tagged with the new epoch ID and the state is buffered in the cache. When a bug is detected, recently executed epochs are squashed and execution rolled back to a point before the bug. Bug characterization can then proceed.

To support this new scheme, we need to modify TLS's mechanism of committing epochs. Roughly speaking, in plain TLS, epochs commit when they can no longer violate sequential semantics. In our scheme, epochs execute in sequence and, therefore, can never violate sequential semantics. However, we do not want to commit them eagerly. Instead, epochs commit only when one of two special events forces them to. One such event is a cache conflict where space is needed in a set where all lines are uncommitted. The other event is when the processor exceeds *MaxEpochs*, the maximum number of uncommitted epochs it is allowed to have. In the case of a cache conflict, the epoch that owns the line to be displaced and its predecessors are forced to commit. Note that this new commit policy does not affect forward progress, since the buffering of epochs that can no longer cause violations is strictly a best-effort approach — our scheme never prevents a displacement that would be allowed by plain TLS.

With this scheme, a given cache hierarchy will frequently need to contain multiple versions of the same cache line, each one tagged with a different epoch ID. This is because epochs belonging to a single thread of execution are likely to access overlapping addresses. If we had to commit epochs simply because their accesses overlap, we would significantly reduce our rollback capability. Fortunately, support for multiple versions of the same line in a cache is provided by plain TLS (Section 3.1.1). Section 5.3 discusses some implementation issues.

3.3. Adapting TLS for Deterministic Re-Execution

The mechanisms for epoch order monitoring and enforcement present in TLS can be reused to order the actions from different threads in multithreaded code. Such order is required to support deterministic re-execution of the buggy code.

In plain TLS, epochs executing on different processors have a total order, given by the sequential semantics of the program being parallelized. To monitor and enforce epoch ordering, coherence requests carry the ID of the originating epoch. In a cache, the ID of the incoming request is compared to the IDs of the lines with matching addresses. If the coherence action has been performed out of order, epochs are squashed to enforce order.

In multithreaded codes, the epochs that we build within a thread are ordered relative to each other by the sequential semantics of that thread. However, there is no a-priori order between the epochs of different threads. In our new scheme, we use the dynamic flow of memory values to create an order: when a value generated by one epoch is read by another epoch in another thread, we set the first epoch to be a predecessor of the second epoch. The overall result is a partial order between the epochs of a multithreaded program. Section 5.2 describes an implementation of distributed epoch IDs based on logical vector clocks that supports such partial order.

Once two epochs from two different threads have been ordered by communication, plain TLS mechanisms are used to enforce the existing order. Specifically, if these two epochs communicate again, their epoch IDs are compared as in plain TLS and a violation is flagged if the communication is out-of-order. In this case, the successor epoch is squashed and re-executed to maintain order. Note that forward progress is guaranteed by the absence of cycles in the partial order of epochs. Indeed, to create a cycle would require ordering epoch A to precede epoch B when B already precedes A. This cannot happen because new epoch ordering is only introduced when two unordered epochs communicate for the first time.

With this support, deterministic re-execution after rollback is only a matter of re-executing the same epochs using the order observed in the first execution. All reads get exactly the same data

as in the first execution, so each epoch performs exactly the same computation and writes to memory as in the first execution.

3.4. Epoch Size Tradeoffs

To enhance debugging, we want to support a large *Rollback Window*, which is the number of dynamic instructions per thread that can be rolled back. However, we also want to minimize the hardware support required and the slowdown caused to the application. An important design decision that affects these parameters is the epoch size.

If epochs are small, both application slowdown and hardware requirements increase, without actually delivering a large Rollback Window. Indeed, in this case, we need many epochs to maintain even a modest Rollback Window size. These many small epochs slow down the application with the overhead of frequent copying to save the architectural registers. Moreover, they increase hardware requirements because we need storage for many register copies and wider tags in the cache to maintain the numerous local epoch IDs.

Even with all these costs, the Rollback Window is hard to expand. The reason is that all these epochs frequently create multiple versions of the same line (Section 3.2). All the versions of a given line map into a single set of the cache. Given that a cache has a limited associativity, only a limited number of uncommitted versions of a given line can remain in the cache. As a result, some epochs simply have to commit to allow a displacement (Section 3.2) and, therefore, the Rollback Window shrinks. Consequently, its size remains modest.

Having sizable epochs eliminates these problems. However, having few, very large epochs is not optimal either. Committing a large epoch results in having little rollback capability left at that point. Therefore, with large epochs, the size of the Rollback Window oscillates widely, which is undesirable.

To control epoch size, we set a limit on the footprint of the data that an epoch can access before it is terminated. The implementation of this threshold is discussed in Section 5.1.

In addition to the size of the epoch, another important design decision is the product of epoch size times the allowed number of uncommitted epochs. If this product is so large that uncommitted epochs use a sizable fraction of the cache storage, the application may slow down. This is because uncommitted epochs create multiple versions of the same line, therefore using cache space inefficiently. As a result, there is less effective space for the working set of the application and the cache miss rate goes up.

This particular problem can be addressed by allowing the state of uncommitted epochs to overflow into a special area in main memory. Support for such overflow area has been proposed for TLS [19] and can be reused here. However, in this initial study, we choose to keep all uncommitted state in the caches for simplicity.

3.5. Synchronization-Induced Epoch Ordering

If the only criterion that we use to decide when to terminate an epoch is the size of its data footprint, our system can livelock programs. This may occur when threads synchronize, depending on the order of thread arrival. As an example, Figure 1-(a) shows two threads that synchronize on a flag. Consider the case when the consumer arrives before the producer. Assuming that no prior ordering exists between the current epochs in the producer and consumer threads, our system orders a spinning epoch in the consumer thread *before* an epoch that sets the flag in the producer thread. From this point on, the TLS hardware enforces this order. As a result, the spinning epoch keeps getting the old value of the flag because the new value has been generated by a successor epoch. The result is that the spinning epoch spins forever.

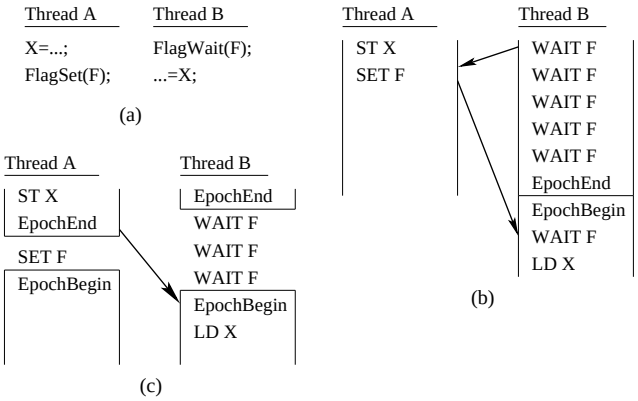


Figure 1. Example code of two threads synchronizing (a), and the results of our optimizations to eliminate livelock (b) and to eliminate unnecessary thread spinning (c). The arrows show the ordering of epochs across threads from predecessor to successor.

3.5.1. Eliminating Livelocks. To address this problem, we add a second criterion to terminate an epoch: when it has executed a certain maximum number of instructions. With this extra support, the spinning epoch in the example eventually terminates. Then, the new epoch in that thread reads the flag again, is ordered as a *successor* of the epoch that set the flag, gets the updated value, and ends the spinning (Figure 1-(b)). Overall, with this support, we eliminate the possibility of livelocks.

3.5.2. Eliminating Synchronization Delays. The approach described above still has a drawback: when threads synchronize, depending on the order of thread arrival, a thread may spin for longer than necessary. The result is a slower program execution.

To eliminate this overhead, we force epochs to terminate when they synchronize. Moreover, synchronization operations are performed using the plain coherent memory accesses that the processors support by default, rather than using TLS accesses. Finally, after synchronization, a new epoch starts. With this extra support, threads do not spin unnecessarily. For example, Figure 1-(c) shows the result of the optimization: since the spinning on the flag uses plain coherent accesses, the flag update is observed as soon as it occurs, and the consumer thread can proceed immediately. The arrow in Figure 1-(c) shows the ordering introduced between epochs of different threads by these flag synchronization operations.

Figure 2 shows how different synchronization operations order epochs across threads. Of course, epochs from the same thread are ordered by the sequential execution of that thread. Furthermore, epoch ordering is transitive. The result is again a partial ordering of the epochs in a multithreaded program.

To implement the ordering introduced by synchronization, we modify the macros or libraries that implement synchronization op-

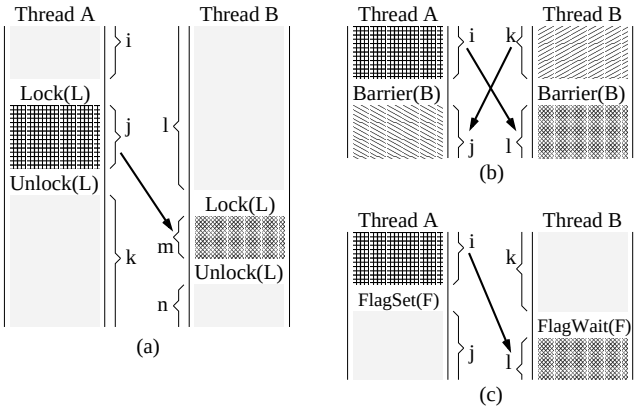


Figure 2. Epoch ordering introduced by lock (a), barrier (b), and flag (c) synchronization. Letters *i*, *j*, ... *n* represent epochs, while the arrows show the introduced ordering of epochs from different threads.

erations (e.g. ANL macros or the pthreads library for POSIX-compliant codes). In addition to synchronizing, these macros or libraries now end the epoch, transfer order information between epochs on different threads, and start a new epoch. To transfer ordering, each synchronization variable has some storage to hold epoch IDs: a single ID in locks and flags, and *N* IDs in barriers. Epochs performing release-type synchronization operations write their IDs; epochs performing acquire-type operations read those IDs and set their own IDs to be *successors* of the releasing epochs. More specifically, in a flag, the producer thread writes its epoch ID before setting the flag; the consumer thread reads that ID only after finding the flag set. For a lock, the current owner thread writes its epoch ID before releasing the lock; the next owner thread reads the ID only after acquiring the lock. For a barrier, arriving threads write their epoch IDs before incrementing the counter; departing threads read all *N* IDs only when they are ready to depart. Overall, with this approach, applications only need to be re-linked with the new libraries to benefit from our epoch-ordering optimization.

In addition to terminating an epoch on synchronization, we still need to terminate it when its data footprint reaches a certain maximum size, and when it has executed a certain maximum number of instructions. The first condition is necessary for programs where threads synchronize infrequently. The second one is necessary for programs where, for example, threads use plain variables to synchronize. In this case, the instruction count threshold is still needed to prevent livelocks as in Section 3.5.1.

4. ReEnact: Debugging Data Races with TLS

While the framework just described can be used to debug many different classes of bugs, we now demonstrate it for one class of bugs: data races in multithreaded programs. Data race bugs are important for two reasons. First, they are notoriously difficult to reproduce and debug. Second, our framework is already capable of *detecting* them. For other classes of bugs, bug-specific detection mechanisms need to be provided. In the following, the application of our framework to race debugging is called *ReEnact*.

In this section, we describe the detection, characterization, and repair of race bugs in ReEnact. After that, we consider how to change ReEnact to debug other classes of bugs.

4.1. Detecting Data Races

Two accesses in a parallel program are conflicting if they are to the same location and at least one is a store [23]. In most programs, conflicting accesses are ordered by well-defined synchronization operations. When they are not, a data race occurs. In this case, program execution has some unintended non-determinism that is typically considered a bug.

Since ReEnact orders epochs using the synchronization operations in the program (Section 3.5.2), in the absence of data races, all cross-thread data sharing happens between epochs that are already ordered. If a race is present, instead, it will appear as communication between epochs that are still unordered. Recall that, when an external request is received by a cache, plain TLS hardware compares the epoch ID of the incoming request and the epoch ID of the matching cache line (Section 3.1.3). Consequently, data race detection in ReEnact is simple: if and only if the epoch ID comparison finds that the two IDs are *unordered*, then a data race has occurred.

To guarantee the detection of all races, ReEnact should not commit an epoch for as long as any other epoch that is unordered with it is still running. This is because the second epoch could still issue a conflicting access. In practice, the finite buffering capacity of the caches and other hardware limitations (Section 3.2) may force ReEnact to commit epochs earlier. Therefore, not all races will be detected, especially long-distance ones. Fortunately, it can be argued that short-distance races rather than long-distance ones are the true dangerous races. Indeed, random scheduling and other factors are more likely to invert the order of accesses in some executions for short-distance races, making program execution unpredictable. Long-distance races are unlikely to cause unpredictability.

Finally, there are programs that have intended data races. While they make the code hard to maintain, they may be necessary to obtain peak performance. To support this case, we allow the programmer to explicitly mark the accesses that are involved in intended data races. When ReEnact detects such races, it does not trigger debugging actions.

4.2. Characterizing Data Race Bugs

As soon as a race is detected, ReEnact starts the bug characterization phase. Often, a single problem such as a missing barrier causes multiple nearby races. Consequently, the goal of this phase is to uncover the *signature* or full structure of the race or set of nearby races. The signature includes information such as the instructions and memory locations involved in the races, the values of such memory locations and, within each epoch, the number of instructions that separate the accesses involved in races.

At the point of race detection, the only available information is one address and one instruction involved in a race. To generate the signature of the set of races, ReEnact proceeds in two steps. First, it continues program execution for more time to detect more nearby races. In the second step, the epochs in the Rollback Window are undone and re-executed in exactly the same order with additional instrumentation.

In the first step, ReEnact will collect the memory locations involved in any races detected. When a race is detected (including the first one), ReEnact sets the relative order between the two

involved (unordered) epochs as indicated in Section 3.3. To enforce this order, and as per normal TLS, any further races between the same two epochs may cause one of the epochs to be squashed, rolled back to its beginning and re-executed, possibly several times. In this first step, program execution is not allowed to go too far. In particular, when further execution would require committing any of the epochs involved in a race already found, execution stops. At that point, for each race, ReEnact has recorded the address of the participating memory location, and the execution order of the participating epochs.

In the second step, all the epochs not involved in the races that can commit, do so, while the others are rolled back. ReEnact then sets watchpoints at the addresses participating in the races. This can be done, for example, with the Debug registers of Pentium 4, which will stop the program whenever the processor accesses one of the marked addresses [12]. Finally, ReEnact re-executes the rolled-back epochs in the *same order* that was observed and recorded in the first step. Every time that execution stops because the program accesses one of the watchpoints, a ReEnact handler records all the information necessary to build the race signature. If the processor does not have enough Debug registers to generate the complete signature in a single re-rerun, ReEnact can squash and re-execute the Rollback Window several times. In each run, the execution is deterministic.

Our use of exceptions to collect the signature of races requires an extension to plain TLS. In plain TLS, an exception causes speculation to end. In ReEnact, these exception handlers run non-speculatively while the state of the epochs remains buffered. To prevent the displacement of speculative state from the caches, these handlers run uncached.

4.3. Pattern Matching Data Race Bugs

Many common data race bugs have obvious signatures. Consequently, we can further characterize a race bug by comparing its signature against a library of known race patterns. If a match occurs, ReEnact can report the cause of the bug to the programmer with high confidence. Such pattern matching can be done by ReEnact on the fly while debugging.

As a proof of concept, we have created a small library with race patterns caused by common bugs. These bugs include using synchronization operations that are hand-crafted with plain variables, and missing synchronization operations. The former are important to detect because such synchronization operations may exhibit unexpected behavior in modern machines that use relaxed memory consistency models. Figure 3 shows the race bugs considered in the ReEnact library. For each bug, the figure shows an example code snippet (a1 to d1) and the resulting race pattern to be matched (a2 to d2). In the patterns, the arrows represent detected data races.

The library matches two instances of hand-crafted synchronization. The first one is a plain variable used as a flag where the consumer thread arrives first (Figure 3-(a1,a2)). The second one is an all-thread hand-crafted barrier that is built with a critical section protecting a count, and a spin on a plain variable (Figure 3-(b1,b2)).

The library also matches two instances of missing synchronization. The first one is a missing lock/unlock for a simple critical section where threads only read and then write a single conflicting location (Figure 3-(c1,c2)). The second one is a missing all-thread barrier that would separate individual threads writing an address and then reading a different one, or vice-versa. Figure 3-(d1,d2) shows one possible case. Note that some of these patterns also take into account the values of variables causing the races and, in the case of barriers, the number of threads involved in the race. Further details are omitted due to lack of space in this paper.

4.4. Repairing Data Race Bugs

After ReEnact generates the race signature, it can present it to the user or send it to the programmer. The signature likely has enough information for a skilled programmer to repair the bug easily. If, in addition, the signature matches a pattern in the library, ReEnact can tell the programmer the cause of the bug with high confidence.

We also envision two scenarios where the bug could be repaired *automatically* on the fly. The first one is under high-confidence patterns like the ones recognized by our library. In this case, ReEnact could force an ordering of the participating epochs that is both legal and consistent with a repair. As an example, consider the missing lock/unlock case (Figure 3-(c1,c2)). Since ReEnact controls the execution, it can undo the Rollback Window one last time. Then, it can stall Thread B before executing the LD X until Thread A has executed at least the ST X. The code is not modified at all, and this execution is perfectly legal. At the same time, it is consistent with the repair of adding the missing lock/unlock. Of course, this repair only fixes one dynamic instance of this bug; the next time this code is executed, the race(s) will reappear. For the other three patterns in our library, a similar analysis can be made. It remains the subject of future work, however, to find out how generally applicable this approach is.

The second scenario for on-the-fly, automatic repair is if the race signature can be sent in real time to the software vendor, and the latter has a patch ready to correct the bug. The patch could then be sent to the user and installed, and execution resumed.

4.5. Extending ReEnact for Other Bugs

The ideas and mechanisms of ReEnact can be extended to debug other classes of bugs. For each class of bugs, we need a few bug-specific extensions: new bug-detection mechanisms, a new set of heuristics to guide bug characterization so it can gather relevant information, and a new library of bug patterns. However, ReEnact's main support, which is the ability to incrementally roll back and deterministically repeat recent execution, can be largely reused.

5. Implementation Issues

We now briefly consider some implementation issues in ReEnact, namely support for epochs, partially-ordered epoch IDs, and multiple line versions.

5.1. Epochs

Most of the hardware needed in ReEnact has already been proposed to support plain TLS. For example, several TLS schemes support multiple speculative epochs per processor (e.g. [5, 26]). Several TLS schemes have a special instruction used to create a new epoch. When the instruction is executed, there is hardware support to back up the architectural registers and generate the ID

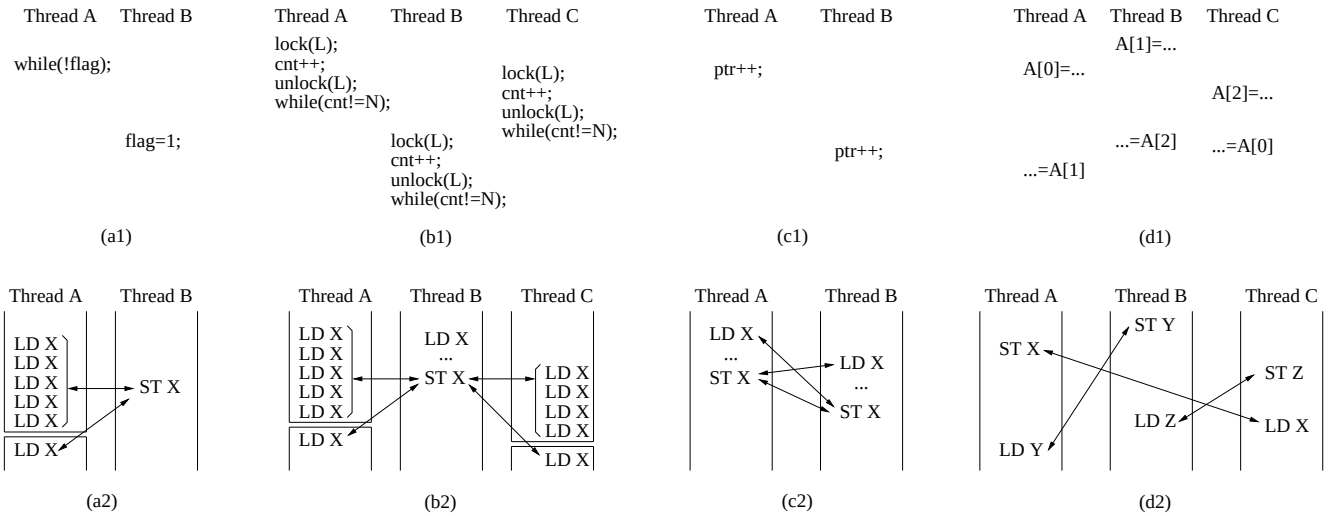


Figure 3. Race bugs considered in the ReEnact library: examples of code snippets with the bug (a1 to d1) and resulting race patterns to be matched (a2 to d2). In the patterns, the arrows represent detected data races.

of the new epoch very quickly and without software intervention (e.g. [5, 10, 13]).

One difference in ReEnact is in the way epochs are terminated and started. An epoch terminates by default when it reaches a synchronization operation; in this case, the next epoch is generated with an epoch-creation instruction. However, as indicated in Section 3.5.2, an epoch is also forced to terminate when it has generated a large data footprint or it has been running for a long time. Consequently, we need two hardware counters: one in the cache that counts the lines that are being accessed by the running epoch for the epoch's first time, and one that counts the instructions executed by the epoch. When one of these counters reaches a threshold count (*MaxSize* and *MaxInst*, respectively), a hardware signal forces a transition in the processor as if it were executing an epoch creation instruction. A new epoch then starts.

5.2. Partially-Ordered Epoch IDs

Partially ordered, distributed epoch IDs are implemented using *logical vector clocks* [7]. This approach has been used in an off-line software-only race detection tool [21]. Each ID is composed of N counters, where N is the number of threads in the system. For example, if we have 4 processors in our chip multiprocessor (Section 6.1), and each counter has 20 bits (allowing up to 2^{20} epochs per thread), we have 80 bits per epoch ID. In our design, we implement them in hardware.

Epoch IDs are manipulated in three situations. First, when we terminate an epoch and start a new one locally, the ID of the latter is made a successor of the former's. Second, the ID(s) of epoch(s) that perform an acquire-type synchronization operation are made successors of the ID(s) of epoch(s) that have performed the most recent release-type synchronization operation on that same variable (Section 3.5.2). Finally, when epochs communicate or commit, their IDs are compared to check whether the IDs are ordered. These three actions are implemented in hardware. They involve simple arithmetic and logic operations as described in [21].

In our implementation, each cache hierarchy has 32 registers to hold the IDs of the local epochs. Each cache line is tagged with an index to these registers. Note that, since the state of committed epochs is merged with the architectural state in main memory lazily, lines from committed epochs linger in the cache. As a result, their epoch-ID registers cannot be freed until all their lines have been displaced from the cache. We limit the number of epoch-ID registers that are needed by providing each L2 cache with a hardware scrubber that, when the number of free epoch-ID registers is too low, traverses the cache in the background and with a low priority. It looks for lines that belong to the oldest committed versions and, when such lines are found, displaces them from the cache. Each pass of the scrubber typically frees a few committed epochs. If, despite of this, a processor runs out of epoch-ID registers, the processor stalls until one is available. However, in our experiments with 32 epoch-ID registers, no such stalls occur.

The ID of the originator epoch is included in every message sent by a processor. In a cache, the ID in an incoming coherence message is compared in hardware against the ID(s) of the matching line(s). To minimize the frequency of these comparisons, it is possible to cache the results of comparing pairs of IDs in a tiny cache, and simply read them out on demand. Most of this hardware support is already present in plain TLS.

5.3. Keeping Multiple Versions

As indicated in Section 3.2, a given cache hierarchy may need to hold multiple versions of the same cache line, each one tagged with a different epoch ID. To reduce the hardware complexity and impact on cycle time, we allow only one (the most recent) version of a line to be in L1. Since the L2 is larger and less critical for performance, it can be designed to hold multiple versions of the same line, at the expense of increasing its access time. Finally, when an epoch finds in the L1 a line belonging to an older epoch, the line is displaced (and written back to L2 if dirty) and a new line is allocated for the new epoch in both L1 and L2. This operation likely induces overhead visible to the processor.

6. Evaluation Setup

To evaluate ReEnact, we use execution-driven simulations with detailed models of out-of-order superscalar processors and their memory subsystems. In the following, we describe the architecture simulated and the applications evaluated.

6.1. Architecture Simulated

The *Baseline* system is a 4-processor state-of-the-art chip multiprocessor. Each of the four processors has two levels of on-chip private caches, and communicates with the other processors through a fast, on-chip 4X4 crossbar network. The caches are kept coherent with a MESI protocol. The chip is connected to main memory with a bus. The details of the Baseline architecture are shown in the top three parts of Table 1. Note that the caches are relatively small because the data sets of the programs are small.

Processor	
Frequency: 3.2 GHz	Integer,FP units: 5,3
Dynamic issue: 6-wide	Pending Ld,St: 16,16
I-window size: 64	Branch penalty: 14 cycles
Reorder buffer size: 128	Branch predictor:
Ld,St,branch units: 2,2,1	Like Alpha 21464
Caches & Network	
L1 size, assoc: 16 KB, 4-way	L1, L2 line size: 64 B
L1 OC,RT: 1,2 cycles	On-chip network: 4X4 crossbar
L2 size,assoc: 128 KB, 8-way	RT to neighbor's L2: 20 cycles
L2 OC,RT: 2,10 cycles	
Front-Side Bus & Memory	
Bus: 400 MHz, 128 bits wide	DRAM bandwidth: 3.2 GB/s
Memory: 2-channel Rambus	Main memory RT: 79 ns
ReEnact Parameters	
Threads/processor: 1	*MaxSize*: Varies (2KB-16KB)
Epoch-ID registers/processor: 32	*MaxInst*: 65,536
MaxEpochs: 2,4, or 8	New L1 version: 2 cycles
Epoch creation: 30 cycles	Any L2 access: +2 cycles
Epoch-ID size: 80 bits	

Table 1. Simulated architecture. In the table, OC stands for occupancy and RT for minimum-latency round trip from the processor. All cycle counts are in processor cycles.

Execution under ReEnact is modeled using the same baseline architecture, except that the two-level cache hierarchy of a processor can hold data from multiple epochs. The last part of Table 1 shows the values of the parameter used — refer to Section 3.2 and Section 5.1 and for their definition. We perform experiments with different parameter values. For example, we vary *MaxEpochs* and *MaxSize*. In our simulations, we model the overhead of epoch creation with a fixed 30-cycle penalty. This penalty includes hardware-based register checkpointing and epoch-ID generation.

As indicated in Section 5.3, the L1 contains only a single version of any line. Consequently, we do not penalize its access time. However, if we need to displace an old version from L1 to create a new version of the line, we charge 2 extra cycles. As for the L2, we assume that the complexity involved in holding multiple versions increases its access time by two cycles over Baseline.

In most aspects, execution under ReEnact proceeds like under ordinary TLS. One major difference is in the treatment of displacements of uncommitted data from the L2 cache. In ordinary TLS, cache lines containing data accessed by an uncommitted epoch cannot be displaced; otherwise program correctness could be compromised. Consequently, we choose the victim for line replacement to be a committed line. If no such line exists in the cache set, the processor stalls until the epoch that owns the line to be displaced commits.

In ReEnact, we also try to avoid displacing lines containing data accessed by an uncommitted epoch. However, if we have to displace such a line, we immediately commit its epoch and all its predecessor epochs, and displace the line. This can be done because epochs remain uncommitted only to enable race debugging. Of course, committing epochs reduces the size of the code that can be undone.

6.2. Applications Evaluated

We evaluate ReEnact using the 12 SPLASH-2 [28] applications. Table 2 lists the input set used in each of the applications. We do not modify the source code of the applications, except when we introduce bugs for our evaluation in Section 7.3. However, we do modify the implementation of the ANL macros that SPLASH-2 applications use to synchronize. Each macro is extended to end the current epoch, create a new epoch, and introduce the appropriate epoch ordering described in Section 3.5.2. To support such epoch ordering, the software data structures that implement the synchronization variables are modified to also store epoch IDs as discussed in Section 3.5.2. These modifications are minimal.

App.	Input	App.	Input	App.	Input
Barnes	16K	Cholesky	tk25.0	FFT	256K
FMM	16K	LU	512x512	Ocean	130x130
Radiosity	-test	Radix	4M keys	Raytrace	car
Volrend	head	Water-n2	512	Water-sp	512

Table 2. Applications evaluated and their input sets.

7. Evaluation

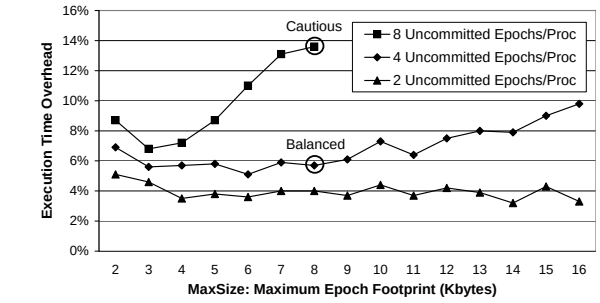
To evaluate ReEnact, we perform three sets of experiments. First, we explore the design space for some key parameters to identify attractive design points. Next, we determine the execution time overhead of ReEnact in race-free execution. Finally, we examine the effectiveness of ReEnact at debugging data races. In all our discussions, the execution time overhead of ReEnact is the additional execution time of the applications on our baseline chip multiprocessor architecture of Section 6.1 when we add support for ReEnact.

7.1. Exploring the Design Space

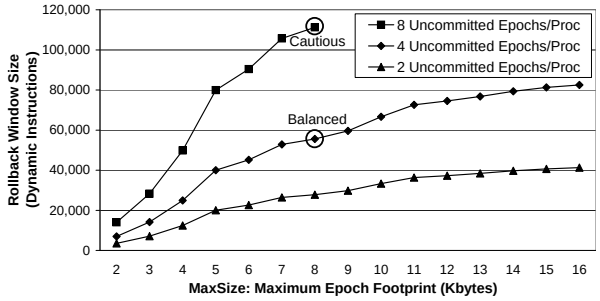
The main design tradeoff in ReEnact is the one between the average size of the Rollback Window and the execution time overhead. Ideally, we want a very large average Rollback Window without slowing down the application much. To select design points in this space, we have two knobs: the maximum number of uncommitted epochs that we allow a processor to simultaneously have (MaxEpochs), and the maximum allowed size of an individual epoch. The second knob can be set by changing the termination thresholds for the epochs: maximum number of instructions executed (MaxInst) and maximum size of the data access footprint (MaxSize). Of these three parameters, we only vary MaxEpochs and MaxSize. MaxInst is set to a large value that does not affect

performance. It can not be infinite because that would create deadlocks in situations described in Section 3.4.

Figures 4-(a) and (b) show the execution time overhead and the size of the Rollback Window, respectively, as we vary our knobs. We vary the maximum number of uncommitted epochs per processor (MaxEpochs) from 2 to 8, and the maximum epoch footprint size (MaxSize) from 2 to 16 Kbytes. The Rollback Window is measured in dynamic instructions per thread. To generate the charts, we compute the average within each application and then across applications.



(a)



(b)

Figure 4. Execution time overhead (a) and size of the Rollback Window (b) for different values of the maximum number of uncommitted epochs per processor (MaxEpochs), and the maximum size of the epoch footprint (MaxSize).

The figures show that, as we increase the maximum number of uncommitted epochs per processor or the maximum size of the epoch footprint, both the Rollback Window size and the execution time overhead generally increase. The main reason for this increase is that uncommitted epochs use an increasingly higher fraction of the cache storage. Recall from Section 3.2 that uncommitted epochs use cache space inefficiently due to replication. Indeed, when two uncommitted epochs access the same memory line, each of them creates its own version. The net effect is less effective space for the working set of the application and, therefore, a higher miss rate. Therefore, we want to limit the number of uncommitted epochs per processor (MaxEpochs) and the maximum epoch size (MaxSize) so that a larger Rollback Window does not come at the expense of unreasonable slowdowns.

The MaxEpochs parameter provides a clear tradeoff between performance degradation and the Rollback Window – more uncommitted epochs per processor allows more rollback, but also results in more replication.

The MaxSize parameter is more difficult to choose. If it is too small, the Rollback Window is small as expected, but the overhead is *increased* due to frequent copying of registers whenever a new epoch begins. By looking at the leftmost part of Figure 4-(a) we see that MaxSize should be at least 4 Kbytes to avoid this unfavorable behavior. Larger sizes of MaxSize provide a tradeoff between performance and Rollback Window size, but there are diminishing returns in Rollback Window size as MaxSize becomes larger. This is due to other limitations on epoch size, the most important of which is that synchronization ends an epoch and begins a new one. When MaxSize is very large, most epochs end because a synchronization is reached and not because MaxSize is reached. Under such circumstances, further increases in MaxSize have little effect on the average epoch size.

As our chosen design point for further evaluation, we select MaxEpochs to be 4 uncommitted epochs per processor, and MaxSize to be 8 Kbytes of buffered state per epoch. We call the chosen design point *Balanced* (B). This configuration has a modest execution time overhead (5.8% on average) and a large Rollback Window (56,000 instructions per processor on average). As a second design point, we select a *Cautious* (C) configuration that supports a large Rollback Window, albeit at a higher performance cost. Such a configuration allows the number of uncommitted epochs per processor to increase to 8. As a result, it suffers a 13.8% average execution time overhead while providing an average Rollback Window of 111,000 instructions per processor. Further increases in Rollback Window size result in performance overhead that is incompatible with our goal of using ReEnact even in production runs.

7.2. Time Overhead in Race-Free Execution

Figure 5 shows the execution time overhead of our *Balanced* and *Cautious* configurations for each of the applications. The overhead is caused by two main sources. The first one is increased stalls due to memory system effects (*Memory*) such as a higher cache miss rate (Section 7.1), higher L1 and L2 hit times (Section 6.1), and slightly higher memory system traffic. The second source is epoch creation (*Creation*), as described in Section 6.1. Note that some of the SPLASH-2 applications have data races, as we discuss in Section 7.3.1. However, these races do not cause incorrect results for any of the program runs that we have performed. Consequently, in this first experiment, ReEnact simply ignores any races upon detection, to emulate race-free execution.

The average overhead in the *Balanced* configuration is 5.8%. This is small enough to make it attractive to use ReEnact on production runs. All the applications except Ocean have overheads lower than 10%. Looking at the sources of overhead, we see that most of the overhead comes from *Memory*. The exception to this is Radiosity, where frequent synchronization results in frequent creation of small epochs and, consequently, high *Creation* overhead.

In the *Balanced* configuration, the maximum size of all the uncommitted epochs per processor is 4 times 8Kbytes, or 32Kbytes. While this is only one quarter of the L2 cache in each processor, the data replication present reduces the space available for the rest of the application working set. As a result, on average for the applications, the L2 miss rate in *Balanced* is 6.2% higher than in the baseline system. This causes most of the overhead in the figure. Ocean has a large working set and, therefore, suffers relatively

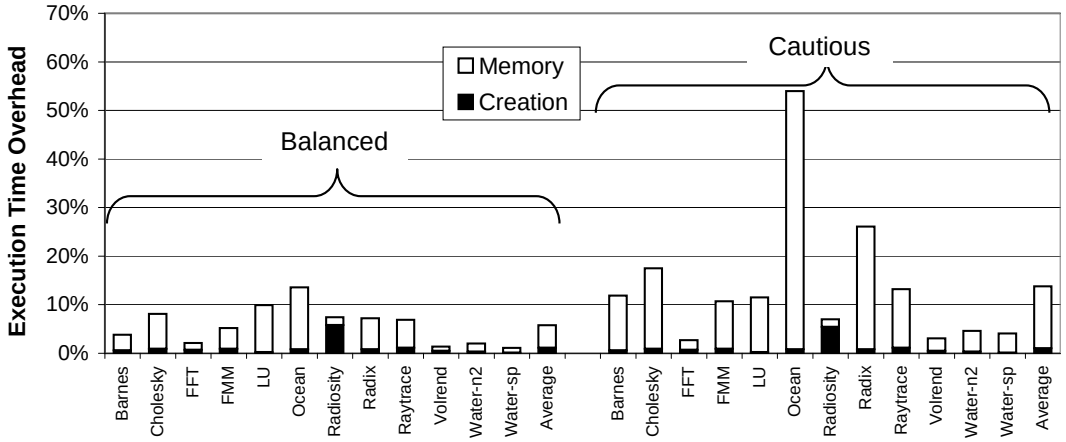


Figure 5. Execution time overhead of the *Balanced* and *Cautious* configurations for each of the applications.

more from reduced cache space. Its L2 miss rate in *Balanced* is 14.7% higher than in the baseline system. This is the reason for Ocean's higher overhead in Figure 5.

For the *Cautious* configuration, the average overhead is 13.8%. In this case, the maximum size of all the uncommitted epochs per processor is 8 times 8Kbytes, which is half the size of the L2. Due to the tighter space in L2, the L2 miss rate is 28.2% higher than in the baseline system. In Ocean, the L2 miss rate is 76.2% higher than in the baseline system. As a result, Ocean's execution time overhead in Figure 5 is 54%. This is too high an overhead to impose on a production run. Consequently, configurations such as the *Cautious* one can only be used in development runs.

7.3. Effectiveness at Debugging Races

To evaluate the effectiveness of ReEnact at debugging data races, we experiment on both applications that originally have races and applications where we insert races. In the experiments, we are interested in five questions: (1) is the race detected?, (2) is the detection early enough that allows ReEnact to roll back execution to a point before the bug occurred?, (3) is the race fully characterized?, (4) does the race signature match one of our race patterns?, and (5) is the race repaired on the fly and application execution successfully completed?

In answering these questions, our sample of experiments is very small: a few hundred dynamic instances of existing bugs and 8 inserted bugs. Consequently, for each of the five questions, we can only assess the effectiveness of ReEnact *qualitatively*. Table 3 shows a summary of our assessment using words like "high" or "medium". We now describe each experiment in detail.

7.3.1. Applications with Existing Bugs. ReEnact detects many dynamic instances of races in several of the SPLASH-2 applications. Specifically, out-of-the-box versions of Barnes, Cholesky, FMM, Ocean, Radiosity, Raytrace and Volrend have races. Overall, we encountered a few hundred such instances. These races do not cause incorrect results for any of the program runs that we have performed. They are caused by hand-crafted synchronization (Section 4.3) and other constructs.

Hand-crafted synchronization occurs often. For example, function *Ray_Trace* in Volrend uses a hand-crafted barrier (Figure 6-(a)). Function *Hackcofm* in Barnes uses a hand-crafted flag called

```
...
LOCK(Global->CountLock);
Global->Counter--;
UNLOCK(Global->CountLock);
while(Global->Counter);
...
```
(a)

```
...
Done(l)=TRUE;
...
...
while(!Done(r));
...
Done(r)=FALSE;
...
```
(b)

```
...
LOCK(Gl->lock_array[b->parent->exp_lock_index]);
pb->interaction_synch+=1;
UNLOCK(Gl->lock_array[b->parent->exp_lock_index]);
...
...
while(b->interaction_synch!=b->num_children);
...
```
(c)

```
...
LOCK(gl->IndexLock);
ProcID=gl->Index++;
UNLOCK(gl->IndexLock);
...
```
(d)

```
...
INTRAF(&gl->VIR,ProcID);
BARRIER(gl->start,NumProcs);
INTERF(gl->start,&gl->VIR,ProcID);
for(i=1;i<=NSTEP;i++){
...
```
(e)

Figure 6. Code examples of hand-crafted barrier (a), flag (b), and custom (c) synchronization, and of lock (d) and barrier (e) synchronization removed to induce bugs.

Done in each element of the *cell* structure (Figure 6-(b)). In FMM, each element of the *_Box* structure has a hand-crafted synchronization counter called *interaction_synch*, which threads increment and wait on until it becomes equal to *num_children* (Figure 6-(c)).

For hand-crafted synchronization, ReEnact's effectiveness at race detection is very high (Table 3). It detects all instances where the consumer thread arrives first and spins, since it appears as an infinite loop. It also detects many cases when the consumer arrives last and finds the variable set. Execution rollback and race characterization are also very effective. At this point, however, ReEnact only pattern-matches hand-crafted barriers and flags, which have a very clear signature (Section 4.3). Other hand-crafted synchronizations do not match any pattern in our library. For example, the counter in Figure 6-(c) from FMM involves reads and writes that do not match our patterns for hand-crafted barriers or flags (Section 4.3). Consequently, we set the entry for pattern-match in

Experiment	Type of Bug	Detection?	Rollback?	Characterization?	Pattern-Match?	Repair?
Existing bug	Hand-crafted synch	Very high	Very high	Very high	High	High
(Instances: several hundred)	Other	High	Unknown	Unknown	No	No
Induced bug	Missing lock	Very high	Very high	High	High	High
(Instances: 8)	Missing barrier	Very high	Medium	Medium	Medium	Medium

Table 3. Qualitative assessment of the effectiveness of ReEnact.

the first row of Table 3 to only high. As for repair of these races, ReEnact repairs them by introducing epoch ordering as described in Section 4.4. All races that match the patterns in the library are successfully repaired.

There are other constructs that create races in SPLASH-2, such as multiple updates to a single variable by different threads without synchronizing. They are accounted for in the second row of Table 3. Since ReEnact flags many such problems when running SPLASH-2, we rate the detection effectiveness as high. However, we have not examined the effectiveness of rollback or characterization. We do not pattern-match or repair these bugs.

7.3.2. Applications with Induced Bugs. We have performed eight experiments adding race bugs to SPLASH-2 applications. Our experiments consist of removing a single static instance of a lock or a barrier per run. For example, in Water-sp, we remove the lock that protects the assignment of thread-IDs to newly-formed threads at the beginning of the parallel section (Figure 6-(d)). These IDs take the values 0-3, and are used to partition the work to do. Without the lock, the program never completes.

As another example, we remove one barrier at a time in Water-sp. For example, we remove the barrier that separates the initialization into two phases (Figure 6-(e)), or the one that separates initialization and main computation.

In all our missing lock and barrier experiments, ReEnact detects the bug. After detection, however, there are differences between missing locks and barriers. For missing locks, rollback is also very effective because the size of the code that needs to be rolled back is usually modest – critical sections tend to be small. Characterization, pattern-matching, and repair for missing locks are a bit less effective. This is because some missing locks produce slightly complicated race signatures, which include multiple reads and writes per thread to multiple variables. Recall from Section 4.3 that we only pattern-match the simplest race signatures for missing locks. However, all those races that are successfully pattern-matched are also repaired by imposing ordering on the epochs involved as described in Section 4.4.

For missing barriers, rollback is less effective because long-distance rollback is sometimes required. Indeed, consider a load-imbalanced code section followed by a missing-barrier bug. A thread with little load may go past the missing barrier early on. When the other threads arrive at the missing barrier and issue conflicting accesses, we may be able to detect the race. However, by this time, the early thread may have already committed the code around the missing barrier and, as a result, not support rollback to that point. This limitation also makes it hard to characterize the bug as a missing barrier, and the pattern-matching fails. This problem occurs in every instance of the missing-barrier bug for the *Balanced* configuration. For the *Cautions* configuration, rollback succeeds in some cases, which are then characterized, pattern-matched, and repaired as usual. Thus, we rate the effectiveness of these steps as medium.

7.4. Discussion

ReEnact's main goal is to effectively detect, roll back, and fully characterize race bugs on the fly while, in race-free conditions, induce overhead small enough to be compatible with production runs. The data in this section has shown that the goal is attained. Highly-accurate pattern matching and subsequent automatic repair, while important, are more elusive targets that may need artificial intelligence algorithms. In any case, after the race is fully characterized, a skilled programmer can likely fix it quickly.

We have not reported execution time overhead for the runs where races are detected and characterized. The reason is that our simulator and debugging handler infrastructure are not streamlined enough to provide representative overhead numbers for characterization and repair code.

8. Related Work

Data races are a well-known source of problems in multi-threaded programs and much work has been done to address them. Previous work on data race detection includes software implementations of race detectors (e.g. [3, 6, 18, 21, 22]), hardware-based proposals to detect violations of consistency models [9] and theoretical work (e.g. [1, 4, 15]).

The work most related to ours is RecPlay by Ronsse and De Bosschere [21], which describes a multi-pass race debugging tool that uses software instrumentation to detect data races and record the ordering of execution for deterministic replay. However, the overheads of data race detection in RecPlay are too expensive for always-on use in production runs: the execution times are 36.3 *times* longer than in uninstrumented execution. Furthermore, RecPlay provides no rollback mechanism, so each re-execution in the bug characterization phase starts from the beginning of the entire program. In contrast, ReEnact provides data race detection, buffering for incremental rollback and recording of the execution order with the overall performance degradation of only 5.8%. Additionally, unlike RecPlay, ReEnact automates the bug characterization phase. The advantages of RecPlay are that it requires no hardware support, and that it can provide data race detection and deterministic re-execution across the entire execution of the program. ReEnact's limited buffering capabilities confine its data race detection, incremental rollback, and deterministic re-execution to a window of several tens of thousands of dynamic instructions per thread. However, our evaluation indicated that this limited window is very effective at debugging races.

The hardware mechanisms we use for debugging are based on TLS [5, 10, 11, 19, 26, 27]. Recently, Oplinger and Lam [17] have proposed using TLS to execute software assertion checks in parallel with the main program, and to provide transaction-like semantics to the programmer. Software assertion checks can be used to detect many types of bugs, while ReEnact currently only detects data races. However, parallelizing assertion checks as in [17] requires the availability of "free" processors or execution contexts

on which to run them. In reality, multithreaded codes may not leave free processors for this. In contrast, ReEnact does not require extra processors and is fully compatible with multithreaded software. As for the transaction-like blocks provided by Oplinger and Lam, they can be used to recover from errors that the application programmer has anticipated and written handlers for. In contrast, ReEnact provides always-on support for incremental rollback without requiring changes to the application. Additionally, ReEnact provides support for deterministic re-execution and a set of heuristics to automate bug characterization and possibly repair.

9. Conclusions and Future Work

The broad goal of our research is to design hardware support to effectively detect, characterize, and possibly repair software bugs on the fly in production runs. When this is accomplished, the payoff will be a major productivity increase in software debugging. We feel that this paper makes a step in this direction.

We identified TLS as a key technology for software debugging. Specifically, with simple extensions, it supports the ability to roll back the buggy execution and repeat it as many times as necessary until the bug is fully characterized. These incremental re-executions are deterministic even in multithreaded codes. Moreover, this operation can be done automatically on the fly, and is compatible with production runs.

As a specific implementation of a TLS-based debugging framework, we introduced ReEnact. ReEnact targets data races in multithreaded programs. Our experiments using SPLASH-2 applications show that ReEnact is very effective at detecting and characterizing data-race bugs automatically on the fly. This we consider our most valuable contribution. Moreover, in many cases, ReEnact also repairs the bug. Last but not least, ReEnact is fully compatible with production runs: the slowdown of race-free execution is on average only 5.8%.

We are currently following two major research extensions. The first one is to support the debugging of races that require long-distance rollback, such as those that occur in load-imbalanced code that is followed by a missing-barrier bug. A promising solution is to develop architectural support to extend cache buffering into main memory, therefore enabling very large Rollback Windows. The second extension is to expand ReEnact to target other types of bugs besides data races. While the core TLS-based mechanisms in ReEnact will not change, we have to provide different bug detection and bug characterization mechanisms.

References

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *18th Intl. Symp. on Computer Architecture*, pages 234–243, 1991.

[2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *32nd Intl. Symp. on Microarchitecture*, pages 196 –207, 1999.

[3] J.-D. Choi et al. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM SIGPLAN 2002 Conf. on Prog. Lang. Design and Implementation*, pages 258–269, 2002.

[4] J.-D. Choi and S. L. Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 145–154, 1991.

[5] M. Cintra, J. F. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *27th Intl. Symp. on Computer Architecture*, pages 13–24, 2000.

[6] K. D. Cooper et al. The ParaScope Parallel Programming Environment. *Proc. of the IEEE*, 81(2):244–263, 1993.

[7] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):23–33, 1991.

[8] M. Garzaran et al. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *8th Intl. Symp. on High-Performance Computer Architecture*, pages 191–202, 2003.

[9] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *3rd Symp. on Parallel Algorithms and Architectures*, pages 316–326, 1991.

[10] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *4th Intl. Symp. on High-Performance Computer Architecture*, pages 195–205, 1998.

[11] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 58–69, 1998.

[12] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, 2002.

[13] S. W. Keckler et al. Exploiting Fine-Grain Thread-Level Parallelism on the MIT Multi-ALU Processor. In *25th Intl. Symp. on Computer Architecture*, pages 306–317, 1998.

[14] E. Marcus and H. Stern. *Blueprints for High Availability*. John Willey & Sons, 2000.

[15] S. L. Min and J.-D. Choi. An Efficient Cache-Based Access Anomaly Detection Scheme. In *4th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 235–244, 1991.

[16] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *29th Intl. Symp. on Computer Architecture*, pages 99–110, 2002.

[17] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *10th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 184–196, 2002.

[18] D. Perkovic and P. J. Keleher. A Protocol-Centric Approach to On-the-Fly Race Detection. *IEEE Trans. on Parallel and Distributed Systems*, 11(10):1058–1072, 2000.

[19] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *28th Intl. Symp. on Computer Architecture*, pages 204–215, 2001.

[20] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *29th Intl. Symp. on Computer Architecture*, pages 111–122, 2002.

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

[22] S. Savage et al. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Trans. on Computer Systems*, 15(4):391–411, 1997.

[23] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Prog. Lang. and Systems*, 10(2):282–312, 1988.

[24] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *29th Intl. Symp. on Computer Architecture*, pages 123–134, 2002.

[25] R. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB - The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[26] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *27th Intl. Symp. on Computer Architecture*, pages 1–12, 2000.

[27] J. Y. Tsai et al. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, 1999.

[28] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Intl. Symp. on Computer Architecture*, pages 24–38, 1995.