PRES: Probabilistic Replay with Execution Sketching on Multiprocessors

Soyeon Park and Yuanyuan Zhou
Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA 92093
{sop009,yyzhou}@cs.ucsd.edu

Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H.Lee* and Shan Lu†
Department of Computer Science
University of Illinois at Urbana Champaign, Urbana, IL 61801
{wxiong2,zyin2,kaushik1,kyuhlee,shanlu}@illinois.edu

ABSTRACT

Bug reproduction is critically important for diagnosing a production-run failure. Unfortunately, reproducing a concurrency bug on *multi-processors* (e.g., multi-core) is challenging. Previous techniques either incur large overhead or require new non-trivial hardware extensions.

This paper proposes a novel technique called PRES (*probabilistic replay via execution sketching*) to help reproduce concurrency bugs on multi-processors. It relaxes the past (perhaps idealistic) objective of "reproducing the bug on the first replay attempt" to significantly lower production-run recording overhead. This is achieved by (1) recording only partial execution information (referred to as "sketches") during the production run, and (2) relying on an intelligent replayer during diagnosis time (when performance is less critical) to systematically explore the unrecorded non-deterministic space and reproduce the bug. With only partial information, our replayer may require more than one *coordinated* replay run to reproduce a bug. However, after a bug is reproduced once, PRES can reproduce it every time.

We implemented PRES along with five different execution sketching mechanisms. We evaluated them with 11 representative applications, including 4 servers, 3 desktop/client applications, and 4 scientific/graphics applications, with 13 *real world* concurrency bugs of different types, including atomicity violations, order violations and deadlocks. PRES (with synchronization or system call sketching) significantly lowered the production-run recording overhead of previous approaches (by up to 4416 times), while still reproducing most tested bugs in fewer than 10 replay attempts. Moreover, PRES scaled well with the number of processors; PRES's feedback generation from unsuccessful replays is critical in bug reproduction.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*

General Terms

Reliability

Keywords

Replay, Concurrency bug

1. INTRODUCTION

1.1 Motivation: The Challenge

Concurrency bugs, such as atomicity violations and deadlocks, are some of the most difficult bugs to detect and diagnose due to their non-deterministic nature. Different from sequential bugs whose manifestations usually depend only on inputs and execution environments, concurrency bugs depend also on the interleaving of threads and other timing-related events [3, 16, 21, 26, 30, 39, 41]. Although a concurrency bug may occur less frequently than a sequential one, it can cause serious damage such as data corruption and hangs, or even catastrophes, such as the Northeast Electrical Blackout [31]. Furthermore, the transition to multi-core processors and the pervasiveness of concurrent programming exacerbate the probability of concurrency bugs in production runs.

In improving the quality of concurrent programs, one of the biggest obstacles faced by developers is to reproduce concurrency bugs. A recent study [16] has shown that the time it takes for a concurrency bug to be fixed depends on how quickly programmers can reproduce it for diagnosis.

Deterministic replay has long been proposed for bug reproduction. The typical approach to support deterministic replay is *re-execution*, because almost all instructions and states can be reproduced as long as all possible factors (referred to as non-determinism) that affect the program's execution can be replayed in the same way. One such factor is inputs, including those from keyboards, networks, files, etc. These can be recorded and supplied

*This work was been done when he was at UIUC as a visiting student.
†This work was done when she was at UIUC as a graduate student. Currently she is an assistant professor at the University of Wisconsin Madison.

SOSP'09 October 11–14, 2009, Big Sky, Montana, USA.

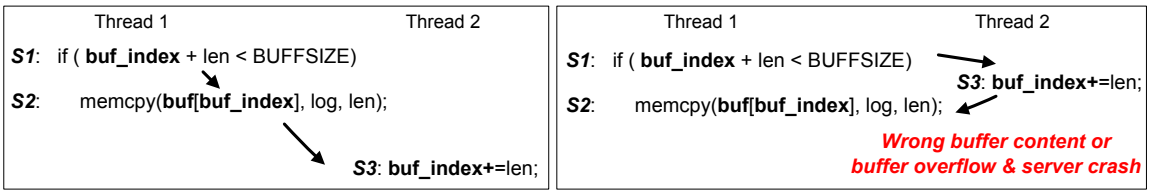

Figure 1: An example simplified from a real Apache concurrency bug. Repeating the bug needs a special input and a special interleaving, which execute S3 between S1 and S2.

during replay [12,32,35]. Another factor is the return values of certain system calls (e.g. gettimeofday). This source can be handled like inputs [12, 32, 35].

The most problematic source of non-determinism comes from timing variations caused by hardware, such as cache state, memory refresh-scrubbing, and timing variations on buses and devices [23,29,39]. Therefore, even when re-executing the same code with the same input on the same machine, the exact time of execution of an instruction or a segment of code could vary from one run to another. Timing variations usually affect only performance, not logic execution and states. However, there are a few exceptions. For example, timing variations can affect *when* a thread is scheduled out, or *when* a thread receives a signal or callback function. To handle these sources of non-determinism, a typical way is to record them with "logic time" instead of physical time. For example, the system can use the number of executed instructions to remember when a thread is scheduled out or when a signal is delivered. Logic time can then be used during replay [12, 32, 35].

Following the above high-level ideas, many systems have been built to provide deterministic replay for *uni-processor* execution. Examples of such systems include the Time-Travel Machine [12], FlashBack [32], VMware [35], R2 [10], and others [2, 5, 20, 37], just to name a few (refer to Section 8 for a detailed discussion).

Unfortunately, on multi-processors (SMPs and multi-cores), besides thread scheduling, signals and asynchronous events, timing variations exert another major influence on a program's execution—how threads concurrently interleave with each other (in addition to those caused by thread scheduling). With multiple threads executing *simultaneously* on different processors, their interleaving can vary from one run to another. Of course, if threads are independent from each other, there is no problem. However, in almost all multi-threaded applications, threads interact with each other via synchronizations and shared memory. Therefore, different interleavings can lead to different variable values and consequently different execution paths as shown in the example below. This makes deterministic replay on multi-processor machines extremely challenging [8, 13, 23, 29, 35, 39], which is why VMware presently supports only uni-processor deterministic replay [35].

Figure 1 shows a real-world bug from the Apache Server. In this example, programmers forgot to protect the pair of accesses to `buf_index`, namely $\{S1, S2\}$, into the *same* atomic region. As a result, Apache crashes when $S3$ is executed between $S1$ and $S2$. This bug is difficult to reproduce, because its manifestation requires not only a special input, but also a special interleaving (i.e., $S3$ executed between $S1$ and $S2$) that rarely occurs during in-house stress testing [26]. In fact, it takes about 22 hours (tens of thousands of iterations) of executions with the bug-triggering input on an 8-core to get reproduced [26]. To diagnose a bug, programmers usually need to reproduce it multiple times to analyze the root cause. If it takes almost one whole day to reproduce the bug one time, the diagnostic process will be excruciating.

1.2 State of the Art

To address the above challenge, several approaches have been proposed for providing deterministic replay for *multi-processors*. The main idea is to record almost every inter-thread interaction such as synchronization and shared memory communication, in addition to those data necessary for uni-processor replay.

Based on the implementation of the above idea, prior work can be grouped into two categories:

(1) Hardware-assisted approaches: Since recording every inter-thread interaction (especially the global order of shared memory accesses) can incur a high overhead, recent work such as Flight Data Recorder [39] and BugNet [23] have proposed new hardware extensions to record such information efficiently. To reduce hardware complexity, various optimizations have been suggested recently, including Strata [21], RTR [41], DMP [6], DeLorean [18], Rerun [11], Capo [19], etc. However, they still require significant hardware modifications, none of which exists today.

(2) Software-only approaches: InstantReplay [14] was one of the first software solutions for deterministic replay on multi-processors. Recently, Strata also has a software-only implementation [22]. Almost all of these state-of-the-art software solutions impose more than 10X–100X production-run overhead, making them impractical [6]. The overhead comes mainly from capturing the global orders of shared memory accesses.

To address the overhead problem, SMP-Revirt [8] recently made very clever use of page protection (instead of instrumenting every shared memory access) to capture shared memory interactions among threads. Due to the page-level granularity, this method works well for applications with coarse-grained data sharing, such as some regular matrix applications. Unfortunately, for applications that have much finer-grained data sharing and more false sharing (such as server applications), this method still imposes 10X or more overhead on 2 or 4 processors [8]. Furthermore, it also has a scalability issue. For example, as reported in the paper [8], FMM's relative overhead increases from 50% to 636% when the number of processors increases from 2 to 4. This is a result of increased false sharing and page contention [1, 9], just like in page-based software distributed shared memory systems [15].

In another recent work, Kendo [25], applies deterministic execution to multi-processors for applications that are perfectly synchronized using locks. For applications that do not satisfy this constraint, including the SPLASH-2 benchmarks selected and evaluated by the authors, programmers need to modify the code before utilizing the tool. This significantly limits its usefulness in practice (especially for legacy code) because many concurrency bugs occur exactly because programmers fail to synchronize correctly.

In this paper, we focus on *software-only* solutions, because hardware modification always comes with a high cost of fabrication and increased hardware complexity.

1.3 Our Observation and Main Idea

One interesting (and perhaps controversial) observation we made is that all of the above work attempts to reproduce the bug on the *first* replay run. As a result, it comes with a prohibitively high cost (10–100X slowdown) on the production run, which is too expensive to be practical.

Compared to a programmers' in-house diagnosis, a production run is much more performance critical because end-users do not want to suffer from developers' bad programming. In addition, since most production runs are bug-free, we do not want to penalize them for bugs that have not happened yet. Therefore, it is important to reduce the production run overhead, even if it comes with slightly increased bug-reproduction time during diagnosis.

This motivation leads to the following questions:

> *Do we have to reproduce a bug on the first replay attempt? Will programmers be happy if the bug can be reproduced within 5–50 replay attempts (especially if it comes with the benefit of significantly lower recording overhead during production runs)?*

To answer these questions, let us look at how developers reproduce a concurrency bug now. Since not much execution ordering/interleaving information is available from production runs, developers simply rerun the program over and over again in the hope that the same concurrency bug will eventually manifest. Unfortunately, in many cases the chance for its reproduction is slim [26]. Many concurrency bugs are not reproduced even after thousands of replays (shown in our experimental results).

In other words, in practice, programmers do not aim to reproduce a bug on the first replay attempt. Therefore, a natural question is: is there anything in between the "reproducing it after thousands of reruns" reality and the "reproducing it on the first replay" goal that existing research ideas are shooting for? Particularly, are there any methods that can reproduce bugs on multiprocessors with only a few (e.g., < 50) replay attempts while adding only small (e.g., $< 50\%$) recording overhead during production run? This is exactly the main objective of this paper, as depicted in Figure 2.

If a method can significantly lower the production-run recording overhead, programmers will probably be willing to tolerate a few extra replay attempts during diagnosis to reproduce an occurred bug, especially when these replay attempts can be automated.

Moreover, the above trade-off becomes even more appealing if this method can guarantee that, once a bug is reproduced successfully (perhaps after several unsuccessful replay attempts), it can then be reliably reproduced in every subsequent replay. This is important to programmers, who usually need to replay a bug many times to identify the root cause.

Additionally, as long as the occurred bug can be reproduced, it is less important for programmers to reproduce it in the exact same way every time. In other words, *even if some of the intermediate, unrelated execution paths differ slightly, it is acceptable, as long as the occurred bug can be reproduced.* In practice, to shorten the diagnosis time, programmers even try to find the minimum triggering conditions needed to reproduce the bug more easily [34].

For convenience of explanation, we refer to the possibility of reproducing a bug in one replay as *"reproduction probability."*

1.4 Our Contributions

Based on the observation above, this paper proposes a novel idea called **PRES (probabilistic replay via execution sketching)** to help reproducing concurrency bugs on multi-processors. It relaxes the previous (perhaps idealistic) objective of "reproducing the bug on the first replay attempt" for the purpose of lowering production-run recording overhead[1]. PRES accomplishes the goal by (1) recording only *partial execution information* during the production run; and (2) using an intelligent partial-information based replayer that systematically explores the unrecorded non-deterministic space via multiple coordinated replay attempts to reconstruct the complete information necessary for bug reproduction. *After a bug is reproduced successfully once after several replay attempts, PRES can then reproduce it with 100% probability on every subsequent replay for diagnostic purposes*.

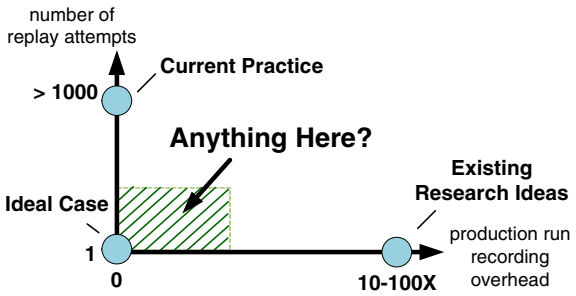

Figure 2: Our objective: exploring the space between existing software-only research ideas and common practice.

Specifically, PRES combines three novel techniques:

(1) **Sketch recording during production run**. Unlike full recording (i.e., recording every inter-thread interaction), *sketch recording* records only important events to minimize the recording overhead. *Even though it may not provide enough information to replay the program in the exact same way as the production run, it does keep the replay close to the production run, as shown in our experimental results*.

This is similar to a "sketch"—an artist frequently draws a sketch first, before putting in details; or to the fairy tale of *Hansel and Gretel*, who leave pebbles or breadcrumbs along the path (at important locations) in order to find their way back home.

In our work, we have explored five different methods of sketch recording[2], including (1) SYNC (recording synchronization global orders), (2) SYS (recording the global orders of synchronization and system calls), (3) FUNC (recording function call global orders), (4) BB (recording basic block global orders), and (5) BB-N (recording every N-th basic block global order). These methods trade reproduction probability for lowered recording overhead at different levels. For comparison with the proposed sketching methods, we also implement another scheme, referred to as RW, which records the global orders of shared accesses to the same memory location. RW represents the previous software-only multi-processor deterministic replay systems [5, 22].

(2) **Partial-Information based Replay (PI-Replay)**. Unlike traditional replayers that have complete information, our replayer (referred to as PI-Replayer) has only partial information (i.e., execution sketches) from a production run. Therefore, our replayer needs to *reconstruct* the *unrecorded* non-deterministic information. It does this by automatically, intelligently, and systematically explor-

[1]Even though we motivate the problem from diagnosing production run failures, PRES can also be used to reproduce concurrency bugs occurred during testing.

[2]In every method, we always record all inputs, signals, thread schedules, and return values from certain system calls such as gettimeofday(), etc., which are necessary for replay even on uniprocessors [12, 32].

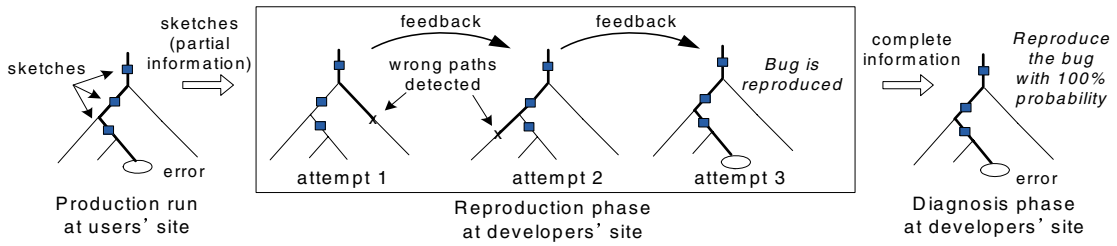

Figure 3: The bug reproducing process of PRES. Bold lines indicate executed paths; gray lines are unexecuted branches.

ing the unrecorded space via multiple replay attempts, and leveraging both sketches and feedback from *unsuccessful replays* (replays that do not reproduce the bug) to finish the "drawing" and reproduce the occurred bug.

(3) **Feedback generation from previous replay attempts**. Unlike traditional replayers that do not record during replay, PRES records all non-deterministic events during replay for feedback generation. Doing so has three benefits: (1) Information recorded from unsuccessful replays can be leveraged to guide subsequent replays to be "smarter." (2) Once a bug is successfully reproduced, the information recorded during this successful replay allows reliable reproduction of the bug at *every* subsequent replay for the programmers' diagnosis. (3) Execution information from unsuccessful replays provides programmers with clues regarding the bug triggering conditions (e.g., why the bug is not triggered under certain global orders?).

Figure 3 shows the steps of PRES:

(1) During the production run, PRES records only important events in an execution "sketch," which is then provided to programmers in case of a failure. Similar to all previous work on deterministic replay, we assume that privacy is not an issue, because either the production run and the diagnosis are both done by the same company (e.g., Google), or there is special agreement or techniques (e.g., [4]) arranged between the customer and vendors.

(2) During the reproduction phase, PRES *automatically* repeats multiple replay attempts until a bug is reproduced. It also reconstructs the complete information necessary for the next phase. After each unsuccessful replay, feedback is provided for subsequent replay attempts.

(3) During the diagnosis phase, PRES leverages the complete execution information from the reproduction phase and reproduces the bug with 100% probability during each replay.

To improve efficiency in searching the unrecorded non-deterministic space, PI-Replayer exploits the following ideas during a reproduction phase:

- *Leveraging execution sketches from production runs to shrink the search space*: The order of many non-deterministic events (e.g., benign races) can be inferred from execution sketches; therefore there is no need to explore alternative orders during a reproduction phase. This filtering process allows us to shrink the unrecorded non-deterministic space in our exploration by up to 500,000 times.
- *Aiming for bug reproduction instead of execution path/states reproduction*: As we discussed briefly, diagnosing a bug does not require reproducing the exact same execution path and states as in the original execution. Therefore, our objective is to reproduce the bug, not the path/states. This relaxation reduces the number of replay attempts for successful bug reproduction by 1.6-5 times, as our experimental results show. Although a successful replay may not be exactly the same as the original execution, it is at least similar. This is because our PI-Replayer leverages execution sketches to monitor every replay attempt, and aborts those which are off-sketch (i.e., producing a different sketch from the one generated by the original execution). Our experimental results show that each time a bug is successfully reproduced, the execution path is only 0.11-0.19% different from the original.
- *Searching space from location closest to failure*: When a replay fails, instead of searching the unrecorded non-deterministic space randomly or from the beginning of the execution, PI-replayer starts from the location closest to the failure. Our experimental results show that this approach reduces the number of replay attempts by up to 6.5 times.

We have implemented PRES using the binary instrumentation tool Pin [17]. We evaluated PRES on 8-core machines with 11 applications of various types including 4 servers (MySQL, Apache, OpenLDAP, Cherokee), 3 desktop applications (Mozilla, PBZip2, Transmission), and 4 scientific/graphics applications from SPLASH-2 [38] (Barnes, Radiosity, FMM, LU). We used 13 representative *real-world* concurrency bugs of different types: atomicity violations, order violations, and deadlocks; data races and non data-races; single and multiple variables. We obtained the following results:

- PRES, with SYNC and SYS sketching schemes, imposed low (< 20%) recording overhead on most evaluated applications, and could successfully reproduce 12 out of 13 tested bugs, most within 10 replay runs. The FUNC and BB-N sketching schemes could reproduce all tested bugs with slightly higher overhead.
- The above low-overhead but coarse-grained recording schemes were capable of reproducing concurrency bugs on multi-processors because our intelligent PI-Replayer could leverage feedback generated from unsuccessful replays. This technique significantly reduced the number of replay attempts needed for bug reproduction. **Without feedback, SYNC and SYS could only reproduce 4 of 13 bugs within 1000 attempts (the maximum limit of replay attempts in our experiments).**
- PRES was scalable across multiple processors. Specifically, SYNC's and SYS's overheads remained small across different configurations (2, 4, and 8-cores).
- Compared to atomicity or order violation bugs, deadlocks usually required less production-run information to reproduce. SYNC sketch reproduced three evaluated deadlocks on the first replay attempt, while adding only small (7%, 15%, and 33%) overhead during the production run.

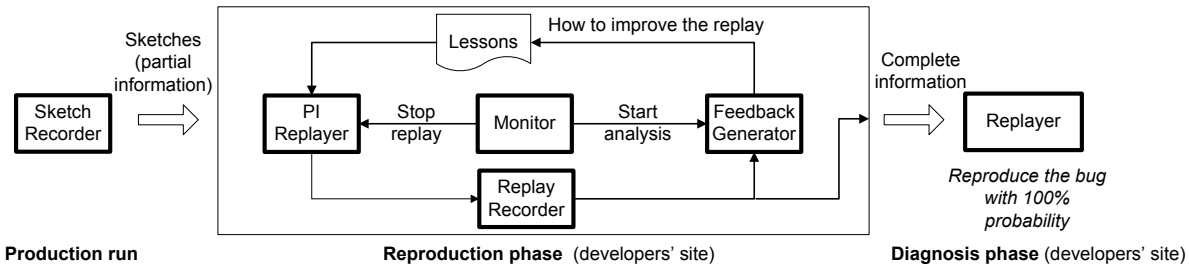

Figure 4: The components and process flow of PRES system

2. BACKGROUND

The starting point of a deterministic replay In order to faithfully reproduce a bug, a replay either starts from the beginning of the program (if the execution is short) or a previous checkpoint. The checkpointing technique has been thoroughly studied and widely used in previous work [8,12,32,34], which shows that the overhead (both space and time) of checkpointing is negligible or very small, especially with infrequent checkpoint.

Our PRES system supports the both ways: replaying either from the beginning of a program, or from a previous checkpoint. The details of checkpointing systems can be referred to in previous work [8, 12, 32, 34].

Source of non-determinism Replay for bug reproduction usually needs to reproduce non-deterministic events to ensure that the re-execution is the same as the original execution. Some non-deterministic events are common to both uni-processor and multi-processor machines. These include I/Os, thread scheduling, system call return values (e.g., timers), signals, etc. Although non-trivial, these events can be handled with less than 10% overhead as demonstrated by previous studies on uni-processor replay [12,32]. Due to space limitation, the rest of this paper does not explain further on how to handle these events, although our implementation deals with all of them.

Multi-processor machines have two extra sources of non-determinism. The first is the order of *concurrent* accesses (with at least one write) to the same location by different threads that execute simultaneously on different processors. For example, as shown in Figure 1, there is a data race on `buf_index`. Concurrent execution of the two threads can generate different values of `buf_index`, depending on how these statements are interleaved. Since all threads are executing concurrently on *different* processors, simply recording thread schedule is not enough for deterministic replay. While many races are bugs (that is probably why programmers need deterministic replay to reproduce them for diagnosis), some are benign and intended for various reasons [24, 40].

The second source is synchronizations (e.g., locks). If two concurrent threads on different processors are competing for the same lock, who gets the lock first is non-deterministic. Therefore, to ensure deterministic replay, it is necessary to record the order of lock acquisitions so that the same order can be enforced during replay. Synchronizations implemented via shared variables (e.g., flags) can be taken care of by handling the first source.

Previous work on deterministic replay for multi-processor records almost all these sources of non-determinism, including interleaving of conflicting shared memory accesses, which can add a significant overhead because it has to monitor and log almost every shared memory access.

3. PRES OVERVIEW

3.1 PRES Architecture and Process

As illustrated in Figure 3 in Introduction and Figure 4 above, our PRES framework includes several modules for dealing with the following three phases: *(1) Phase 1—Production Run:* A sketch recorder records only important events to provide partial execution information for possible off-line replay in case of a failure. *(2) Phase 2—Bug Reproduction:* Four modules (Partial Information-based (PI) Replayer, replay-recorder, monitor, and feedback generator) are used to systematically explore the unrecorded non-deterministic space and reproduce the bug. They also provide the complete non-deterministic information for the next phase needed to reliably reproduce the occurred bug with 100% probability. *(3) Phase 3—Bug Diagnosis:* A deterministic replayer uses the complete information from phase 2 to reproduce the occurred bug reliably for diagnosis. The following discussion briefly describes the modules unique to PRES.

Sketch recorders only record important events. Therefore, there could be many design choices. In order to effectively explore the design space, we consider two factors: (1) recording overhead; (2) usefulness for replay in case of failures. Section 4 will present five different methods of execution sketching.

The partial Information-based Replayer (PI-Replayer) is used in the bug reproduction phase. During each replay, at every non-deterministic point such as synchronizations and shared memory accesses, it consults two information sources: (1) execution sketches collected from the production run and (2) feedback from the previous unsuccessful replay attempts. If information is provided by the former, PI-Replayer passively follows the production-run outcome. If it is provided by the latter, PI-Replayer will decide on an alternative choice for exploration. If no information is available, it is free to execute in any way. See more details in Section 5.

The replay recorder is a heavy-weight recorder that logs detailed information of each replay. This is similar to full-information recorders in previous work [5, 22]. The only difference is that our recording is used only during the off-line bug reproduction phase which is not performance critical, instead of during the production run. It provides three benefits as discussed in the Introduction, namely, (1) feedback to help subsequent replays to succeed; (2) complete information for a diagnosis phase to do a 100% deterministic replay; and (3) clues to programmers regarding the nature and triggering conditions of the occurred bug.

The monitor tracks a replay execution and stops a replay attempt when (1) the replay-run sketch deviates from the execution sketch recorded in the production run, and is therefore highly unlikely to reproduce the bug; or (2) the bug is successfully reproduced. This module is used to improve the efficiency of probabilistic replay: the earlier it stops a hopeless replay and moves on to a new try, the

quicker a production run bug can be reproduced. More details are in Section 5.

The feedback generator analyzes the logs of previous unsuccessful replays and gathers feedback for future replays or programmers' diagnosis. In general, it tries to identify reasons why the bug is not reproduced. This information will be provided to the PI-Replayer to make alternative choices for subsequent replay attempts. Details are in Section 5.

3.2 Information Leveraged by PRES

PRES leverages three types of information: (1) inter-thread global order from the production run, (2) intra-thread local information from the production run, and (3) feedback from previous unsuccessful replays. The first type is used to synchronize threads to repeat the production-run interleaving. Both the first and the second are used by the monitor to check if a replay is still on the right track. The third is generated during replay and is used to reproduce non-deterministic events not recorded during the production run.

4. SKETCH-RECORDERS

The general function of our sketch recorders is to systematically select important operations in the program and record the global execution *order* of them (Figure 5 (a)). We refer to these important operations as *recording points*, unselected operations as *non-recording points*, and the execution between two recording points as a *chunk* (Figure 5 (b)).

Our sketching schemes record the total order of recording points, which sets up a partial order among chunks. Specifically, the execution order between accesses that reside in non-overlapping chunks (e.g., *I1* and *I3* in Figure 5 (b)) is implicitly recorded through recording points. On the other hand, the order between accesses that reside in overlapping chunks (e.g., *I2* and *I3* in Figure 5 (b)) is not recorded. Therefore, in contrast with traditional deterministic replay systems [5, 22], this unrecorded non-deterministic information needs to be intelligently reconstructed by our PI-Replayer for bug reproduction.

Intuitively, the fewer the recording points, the lower the recording overhead during the production run, but more replay attempts may be required to reproduce a bug.

A running example: Figure 6 (a) shows an example that we will use through out this paper. This code snippet is simplified from a SPLASH-2 macro bug. If executed correctly, thread 1 should print out a `result` assigned by the `myid`–0 thread. Unfortunately, certain interleaving, as demonstrated by the arrows in the figure, can cause a wrong output. Specifically, when the child thread (thread 2) acquires the lock first, it obtains `myid`–0 and is in charge of filling the final result. Due to lack of synchronization, the parent thread (thread 1) mistakenly reads the shared variable `result` and prints it out *before* thread 2 updates it. We will use this example to show how different recorders would do the "sketches."

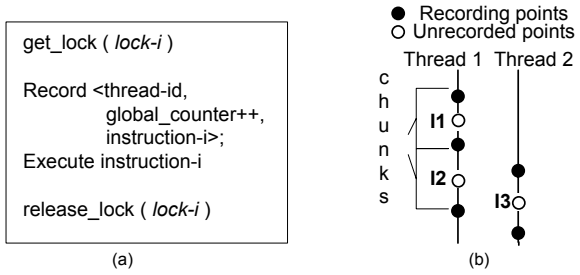

Figure 5: (a) Recording algorithm; (b) Recording points and chunks. `lock-i` is a lock variable. It was used to protect the global counter for the sake of obtaining global order among recording points from different threads. To reduce lock contention, we used a hashed lock for lock-i whenever applicable.

Below we present five sketching methods that trade reproduction probability for recording overhead at different levels, and two basic schemes (Base and RW) that represent two extremes of the trade-off for comparison.

(1) Baseline recorder (Base). Our baseline recorder only records inputs, signals, thread scheduling and other data necessary for deterministic replay on uni-processors [12, 32]. It does not record any global order of events from different threads beyond those implied by thread schedules and asynchronous events.

(2) Synchronization recorder (SYNC). In addition to the information recorded in Base, this scheme records the global order of every synchronization operation. Specifically, instrumentation is put at the *return* of each *lock/unlock* operation (e.g., `pthread_mutex_lock`), conditional wait operation, or other synchronization primitive (note that application-specific synchronizations implemented via shared variables are not recorded). As shown in Figure 6 (b), in this example, only the order among the four lock/unlock operations is recorded.

(3) System call global order recorder (SYS). In addition to synchronization global order, this scheme also records the global *order* of all system calls. Specifically, additional recording points are set right after the *return* of every system call. Thread id, the value of a global order counter, and the name of the invoked system call will be recorded in the log. As shown in Figure 6 (c), in this example, one more recording point is added upon those selected by SYNC.

(4) Function global order recorder (FUNC). [3] This scheme records the global *order* of every function call (entry points or return points). At each recording point, thread-id, global counter, and the name of the function are recorded. To further reduce overhead, we prune those functions that never access shared variables and therefore contain no non-deterministic events. Figure 6 (d) shows the recording points selected by FUNC.

(5) Basic-block global order recorder (BB) This scheme selects recording points at finer granularity, i,e., every basic block (a code block that does not contain any form of jump instructions). Right before the first instruction of each basic block is executed, a global counter, thread id, and the program-counter of that instruction are recorded. BB will inevitably introduce more overhead than the above four schemes. In our prototype, we used program analysis to avoid recording those basic blocks that only accessed private variables. Figure 6 (e) demonstrates BB sketching.

(6) The N-th basic block global order recorder (BB-N) This scheme optimizes BB by selecting every N-th basic block for recording. The purpose of this method is to explore the spectrum of trading reproduction probability for lower overhead.

(7) Shared reads and writes (RW) This scheme is the same as previous software-only multi-processor deterministic replay systems [5, 22], recording the global *order* of accesses to the same shared variables from different threads. Each shared variable has its own order of accesses. There is no need for a global order of all shared accesses. We can apply some transitive-reduction optimizations [39]. However, they can only help reduce the log sizes, not

[3] While the names of SYS, SYNC and FUNC may be easily confused with the R2 idea, they are very different, as discussed later in Section 8.

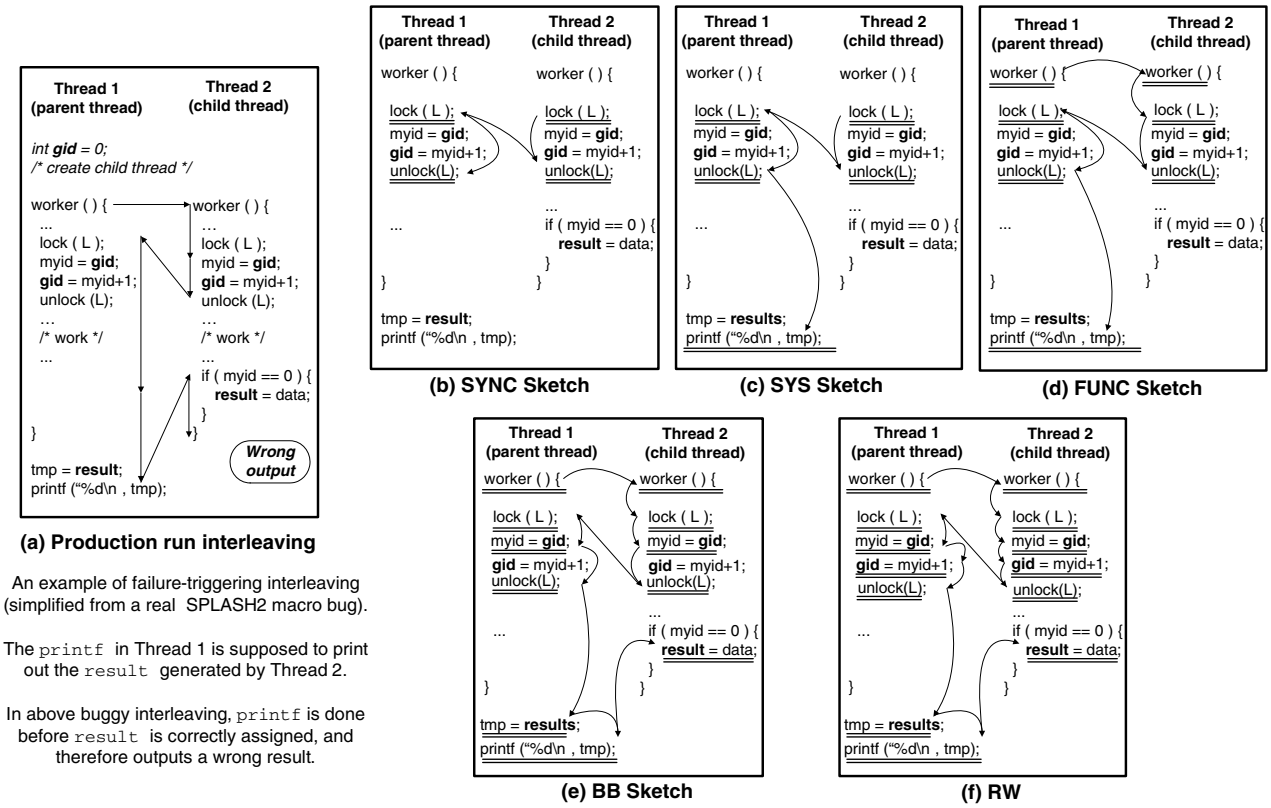

Figure 6: Sketch recording examples. The production run interleaving that caused failure is shown in (a). The `gid` and `result` are shared variables; The `myid` is local. Underlined code in figure (b–f) shows the recording points under different sketches. With each sketch subsuming previous ones, more and more recording points are taken from SYNC to RW.

the recording time overhead, since almost every shared access still needs to be instrumented.

Subsuming relationships Except for BB-N, between any two consecutive methods presented above, there is a subsuming relationship. For example, anything recorded by SYNC would also be recorded by SYS. Similarly, anything from SYS would also be recorded by FUNC. As every function entry or return starts a new basic block, BB subsumes FUNC. Since we only record the orders of basic blocks with at least one shared access, RW also subsumes BB. Such subsuming relationships give us the opportunity to examine the trade-offs between recording overhead and reproduction probability in an incremental spectrum of execution sketches.

Other possible sketching schemes It is certainly conceivable to add other methods, such as every N-th function, the combination of FUNC and BB-N, etc. As a starting point for demonstrating our main idea, we focus on these five methods. We hope that our idea will inspire others to explore more effective methods.

5. PARTIAL INFORMATION-BASED REPLAYER

5.1 Basic PI-Replayer

PI-Replayer periodically consults the execution sketch generated during the production run to maintain the correct execution order among recording points. Specifically, right before a thread executes a recording point, PI-Replayer checks the sketch to make sure that all prior recording points from other threads have been executed.

For operations that are not recorded in the sketch, PI-Replayer is free to execute them in any way. In the next subsection, we will describe how feedback is leveraged from previous unsuccessful replays to guide the execution of such operations.

The **Monitor** module keeps a close eye on each replay for the following two purposes.

(1) Detecting off-sketch executions This task is accomplished by comparing replay run events against execution sketches for inconsistencies. For example, with SYNC sketch, if an extra synchronization is performed that does not match the sketch, the replay is considered off-sketch.

(2) Detecting bug reproduction To detect whether a production-run failure has been successfully reproduced, the PRES Monitor requires information from programmers regarding the failure symptoms. For crash failures, it is straightforward, as PRES can catch exceptions. For incorrect results, it is more complicated. Programmers need to provide conditions just like conditional breakpoints to examine outputs for anomalies. To detect whether a deadlock bug is successfully reproduced, the monitor uses a periodic timer to check for progress. To be conservative, it does not claim a deadlock situation the first time it detects a lack of progress. It waits for a few rounds to see if the program is still stuck in a similar state. PRES can also leverage *testing oracles* that are usually available from regression testing to detect failures, and employ bug detection tools [30, 40] during replay to detect potential bugs in addition to visible failures. Programmers can also use PRES interactively to learn whether a replay has been successful or not.

Note that our monitor does not need to check failure conditions at every instruction. It only needs to perform such a check at every

visible event, such as exceptions, timer signals, and outputs. Therefore, overhead is not an issue, especially since the monitor operates off-line during a bug reproduction phase, instead of during production runs.

5.2 Improve PI-Replayer with Feedback

5.2.1 The cause of unsuccessful replays

Even all sketching schemes other than RW also record almost all non-deterministic events except for the global order of some shared memory accesses. Therefore, the only possible cause for an unsuccessful replay is un-recorded data races. For example, Figure 7 shows an unsuccessful replay with FUNC sketch. Since the execution order between S2 in thread-2 and S1 in thread-1 is not recorded by FUNC, a replay can execute these two instructions in any order. However, if S2 is executed before S1, the bug will not be reproduced.

Therefore, the goal of our feedback generator is to identify race pairs (such as S1 and S2 above) that are responsible for unsuccessful replay attempts, so that subsequent replays can enforce a correct order between them.

Note that feedback generation is critical to bug reproduction, because most bug-triggering execution orders among race instructions have very low probability of occurring without external perturbation, as indicated by [26]. Therefore, if we do not leverage feedback, relying only on random re-execution, it will be very difficult to reproduce concurrency bugs with partial information. This will be further validated in our experimental results (Section 7.2).

Moreover, feedback from unsuccessful replays provides useful clues for understanding the bug triggering conditions for occurred bugs. In the example above, PRES could inform programmers that if S2 executes before S1, the bug will not manifest. This information would help the programmers narrow down the possible causes of the bug.

5.2.2 Replay-time recording for feedback generation

In order to generate feedback, we need to record every non-deterministic event during replay. Since replay is conducted as off-line post-mortem analysis by programmers, it is less performance critical. Therefore, during each replay, we use the RW recording s cheme to record the global order of accesses to each shared variable. The overhead for each replay does not increase much over previous deterministic replay schemes [14], since they also need to instrument shared accesses to enforce the recorded global orders. Moreover, the replay-time recording can be made incrementally since it only needs to record the order of memory accesses in those new execution paths that previous replays did not encounter.

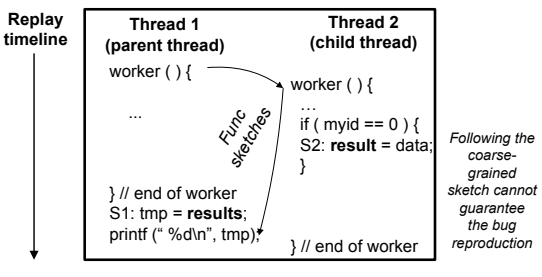

Figure 7: An example of unsuccessful replay. The production run interleaving of this example is shown in Figure 6 (a). The arrow shows the production-run sketch recorded by FUNC.

5.2.3 Generating and leveraging feedback

Our PI-Replayer conducts a repeated and *automated* process, i.e., replaying, generating feedback, using feedback, and replaying again. Specifically, after an unsuccessful replay run, the following steps are conducted:

Step 1: Identifying races. The feedback generator first identifies all dynamic race instruction pairs in the previous unsuccessful replay using an existing race detection algorithm (happens-before algorithm [7,27]). Specifically, our trace analyzer processes the trace of an unsuccessful replay (generated by the replay recorder) and calculates the vector-timestamp (based on lock-operations, thread-creation/join, and barrier operations) for each memory access. It then identifies conflicting instructions whose vector-timestamps are not ordered.

Step 2: Filtering order-determined races. Not every race pair is a suspect for causing the unsuccessful replay. Some race pairs' production-run execution orders are already recorded or implied by the production-run sketch. In other words, we know the exact orders that they should follow, and our replayer always guarantees the correct order in *every* replay. Therefore, they are definitely innocent. We filter them and add remaining race pairs into our suspect set. Specifically, the order of a pair of instructions is implied by the sketch if (1) they are both recording points; or (2) their corresponding chunks are completely ordered in the sketch (e.g., $I1$ and $I3$ in Figure 5 are ordered; $I2$ and $I3$ are not). After the pruning, the remaining race pairs are identified and added to the suspect set. In Section 7, we will show the effect of this filtering process in different sketching schemes.

Step 3: Selecting suspect. From the suspect set, PI-Replayer follows certain policies (described in Section 5.2.4) to flip the execution order of one pair in the next replay.

Step 4: Starting next replay PI-Replayer executes next replays deterministically until the suspect racing pair is reached (a deterministic execution is feasible because every non-deterministic event in the previous replay was recorded). At this point, PI-replayer controls the program to execute the two race instructions in the opposite order from the last run. Once the execution order of this race pair is flipped, the rest of the replay is the same as the default replayer.

The above steps may be repeated many times until the bug is reproduced. The mistakes made by early replays will be corrected, and PI-Replay will gradually get closer to bug reproduction. Since there are a limited number of execution paths and therefore a finite number of races, repeating the above process will eventually find a correct interleaving and successfully reproduce the bug.

5.2.4 Challenging design issues

How to select a suspect race pair for the next replay? Although the main idea of selecting and flipping races is straightforward, in practice it is non-trivial for three reasons. (1) For each replay, more than one race pairs could be identified as suspect. Which one should we select to flip first? (2) If the next replay run still fails, new suspect race pairs could be generated. How shall we prioritize them along with the old ones? (3) It is possible that more than one race pair's execution order needs to be flipped in order to correct an unsuccessful replay. How shall we explore these different combinations?

To address these challenges, PI-Replayer uses a divide and concur strategy. First, it uses a *depth-first* policy to decide which historic, unsuccessful replay to revise for the next replay. Second, from the selected historic replay, PI-Replayer follows a *close-to-failure-first* principle to choose a suspect (unfiltered) race to flip in the next replay.

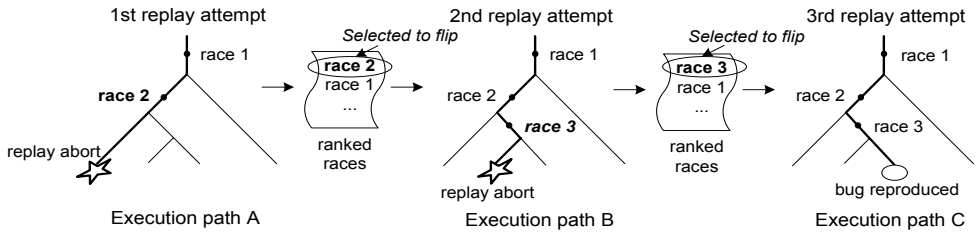

Figure 8: Coordinated replay runs

To implement the above strategy, PI-Replayer uses two stacks, one called *ReplayStack* which records historic replays that are still worth exploring, and the other called *RaceStack* which records data races to be flipped in future replays. When a replay fails, PI-Replayer checks whether this replay has made more progress (i.e., executing more sketching points) than the previous historic replay, and if it has revealed any new suspect races. If so, this replay is pushed into ReplayStack, and newly revealed races are pushed into RaceStack according to their order of occurrence (i.e., the most recently encountered race is inserted last). PI-Replayer also records the association between this replay and the *newly inserted* races.

For the next replay, PI-Replayer follows the first replay on the top of ReplayStack, and flips the race on the top of RaceStack. When PI-Replayer has explored all races associated with (i.e., newly revealed by) a replay (note that it is not all races *encountered* during a replay), this replay is popped out of ReplayStack since it does not provide any unique information for exploration.

Figure 8 shows a simple example of how PI-Replayer explores the unrecorded non-deterministic space based on the above strategy. In Section 7, we will show the quantitative benefit of the above search strategy over alternative ones.

How to guarantee the suspect is always recorded? In order to record both instructions of a suspect race pair, we should not abort an unsuccessful replay too early. In our experiments, we observed a few cases where an off-sketch deviation was detected before the second instruction of the suspect race pair was executed. To handle this problem, when an off-sketch is detected in one thread, PI-Replayer does **not** stop the other threads until they reach the end of their current chunks (i.e., the next recording point). We have theoretical proof that this strategy will address the problem. We omit it due to space limitations.

Transitive-reduction optimization For more efficient replay, PI-Replayer also uses transitive reduction (similar to that in FDR [39]) to avoid picking the race pairs whose execution-order flipping can be automatically achieved by flipping some other race pairs.

6. EVALUATION METHODOLOGY

We implement PRES (including all five sketching methods) using Intel's dynamic instrumentation tool, Pin [17], for recording and also for replay[4]. The bug reproduction phase is entirely automated. The experiments are conducted on an 8-core Intel Xeon machines (2.33GHz, 4GB of memory) and Linux version 2.6.9.

Evaluated replay schemes Our experiments evaluate and compare PRES sketching schemes and three other replay systems including *Base*, *Pin*, and *RW*, as illustrated in Table 1. We use *Pin* to show the overhead introduced by Pin (without any instrumentation).

[4]Our overhead results would be better with static instrumentation tools, but due to the unavailability of static tools that work for multi-threaded server applications, we chose Pin and paid the unnecessary overhead of "dynamic" instrumentation.

PRES schemes	
SYNC	record/replay synchronization (lock) order
SYS	record/replay the global order of all system calls
FUNC	record/replay the global order of every function
BB-2	record/replay the global order of every other basic block
BB-5	record/replay the global order of every five basic block
BB	record/replay the global order of every basic block
Schemes Representing Previous Techniques	
Base	Baseline: no global order recording
Pin	Simply conduct BASE upon the PIN instrumentation framework (base component of all PRES and RW schemes).
RW	record/replay all shared memory accesses

Table 1: Evaluated PRES schemes

Application	Size (LOC)	Application description
Apache	1.9M	Web server
Cherokee	88K	Web server
MySQL	1.9M	Database server
OpenLDAP	0.3M	Directory Access Server
Mozilla	3.4M	Web browser suite
PBZip2	2.0K	Parallel BZip2 file compressor
Transmission	86.4K	BitTorrent client
Barnes	3.0K	Barnes N-Body algorithm (SPLASH-2)
FMM	4.9K	FFT N-Body algorithm (SPLASH-2)
LU	1.0K	LU matrix multiplication (SPLASH-2)
Radiosity	24.5K	Graphics rendering (SPLASH-2)

Table 2: Evaluated applications

Bug Type		ID	Bug description
Atomicity Violation Bugs	Single variable	Apache#1	Non-deterministic log-file corruption
		Cherokee	Non-deterministic buffer corruption
		MySQL#1	Non-deterministic DB log disorder
		Mozilla#1	Wrong results of java-script execution
		PBZip2	Random crash in file decompression
	Multi-variable	Mozilla#2	Unsynchronized accesses to array and array-flag causes crash
Deadlock		OpenLDAP	Contention on two locks causes hang
		Apache#2	Circular wait for a lock and a queue
		MySQL#2	Wrong order of two locks causes hang
Order Violation Bugs		Barnes	Problems in platform-dependent macro (introduced by external macro providers) cause various wrong outputs on execution statistics
		LU	
		Radiosity	
		Transmi -ssion	A shared variable is read before it is properly assigned

Table 3: Evaluated real world concurrency bugs

Evaluated applications and real-world concurrency bugs We use 11 representative multi-thread open-source applications (shown in Table 2), including 4 widely used servers (Apache, Cherokee, MySQL, and OpenLDAP), 3 desktop/client applications (Mozilla,

Applications	Pin	SYNC	SYS	FUNC	BB-5	BB-2	BB	RW
Server Applications (throughput degradation)								
Apache	4.23%	7.05%	10.91%	15.55%	18.18%	19.70%	36.32%	53.24%
MySQL	13.48%	15.48%	15.87%	18.06%	32.59%	37.16%	43.93%	75.35%
Cherokee	7.04%	7.24%	7.39%	7.87%	7.88%	8.06%	8.99%	28.45%
OpenLDAP	19.35%	33.03%	33.58%	38.29%	48.01%	52.64%	59.55%	76.20%
Desktop Applications (overhead)								
Mozilla	32.55%	59.58%	60.63%	83.89%	598.01%	858.86%	1213.90%	3093.59%
PBZip2	17.46%	18.00%	18.07%	18.43%	595.43%	1066.64%	1977.96%	27009.79%
Transmission	5.9%	14.50%	21.33%	32.75%	30.34%	33.98%	41.92%	71.80%
Scientific Applications (overhead)								
Barnes	2.53%	6.50%	6.83%	427.76%	424.27%	1122.343%	2351.02%	28702.26%
Radiosity	16.01%	16.11%	16.19%	779.05%	480.35%	1181.73%	2425.53%	27209.67 %
FMM	8.87%	17.50%	17.50%	47.75%	659.41%	1366.94%	2697.55%	31736.43 %
LU	9.23%	11.01%	12.06%	86.75%	668.80%	1315.89%	2633.13%	31018.02 %

Table 4: Recording overhead of each sketching scheme over bare application execution (without Pin or PRES) on an 8-core. For servers, the overheads are measured as the percentage reduction in throughput using publicly available performance benchmarks. For desktop and scientific applications, overheads are measured as the percentage increment in elapsed time. The "Pin" column shows Pin's overhead without PRES.

Application	Bug Id.	Bug Type	Base	SYNC	SYS	FUNC	BB-5	BB-2	BB	RW
Server Applications	Apache#1	atomicity	NO	96th	28th	7th	8th	7th	1st	1st
	Apache#2	deadlock	NO	1st	1st	1st	1st	1st	1st	1st
	MySQL#1	atomicity	NO	3rd	3rd	2nd	3rd	2nd	1st	1st
	MySQL#2	deadlock	NO	1st	1st	1st	1st	1st	1st	1st
	Cherokee	atomicity	NO	33th	27th	24th	25th	8th	7th	1st
	OpenLDAP	deadlock	NO	1st	1st	1st	1st	1st	1st	1st
Desktop Applications	Mozilla#1	atomicity	NO	1st	1st	1st	1st	1st	1st	1st
	Mozilla#2	multi-var	NO	3rd	3rd	3rd	4th	4th	3rd	1st
	PBZip2	atomicity	NO	3rd	3rd	2nd	4th	3rd	3rd	1st
	Transmission	order	NO	2nd	2nd	2nd	2nd	2nd	2nd	1st
Scientific Applications	Barnes	order	NO	10th	10th	1st	74th	19th	1st	1st
	Radiosity	order	NO	NO	NO	152th	5th	1st	1st	1st
	LU	order	NO	2nd	2nd	1st	2nd	2nd	1st	1st

Table 5: The number of replays to reproduce the bug during the reproduction phase. NO means that the bug was not successfully reproduced within 1000 tries—the maximum limit set in our experiments. "Base" means only recording those events that are necessary for deterministic replay on a uni-processor. Note that after a bug is successfully reproduced in this phase, it can be reproduced on every replay during the diagnostic phase with PRES.

PBZip2, and Transmission), and 4 graphics/scientific applications from SPLASH-2 [38]. These applications cover both I/O-bounded (e.g., Apache, Cherokee, MySQL, OpenLDAP) and CPU-bounded (e.g., SPLASH-2) applications; as well as both lock-synchronization (e.g., all server applications) and barrier-synchronization applications (e.g., LU).

In our experiments, we evaluate how different replay schemes can help reproduce 13 *real-world* concurrency bugs (Table 3) of different types in the 11 applications. All of the bugs have caused production-run failures that were reported by real-world users. In addition, these 13 failures have covered most major types of concurrency bugs (as reported by a recent characteristic study [16]): atomicity violations (both single-variable and *multi-variable* ones), order violations, and deadlock bugs. Some of the atomicity violation and order violation bugs are data race bugs. We follow the same categorization as by Lu et al. [16].

7. EXPERIMENTAL RESULTS

7.1 Overall Results

Table 4 shows the overhead for various sketching schemes. Table 5 shows the number of replays needed to reproduce those real world bugs during a bug reproduction phase. More details about the recording overhead and log sizes will be presented in Section 7.7.

Overall, SYS, SYNC and FUNC are all reasonably good. For performance-critical applications, SYS and SYNC should be the way to go since they have small overheads (6-60% overhead for non-server applications, 7–33% throughput degradation for servers) and can reproduce 12 out of the 13 evaluated bugs within mostly fewer than 10 replay attempts. Comparing to RW (which is similar to previous approaches), the performance improvement is huge: 3–4416 times overhead reduction for non-servers and up to 3 times higher throughput for servers.

For applications that are less sensitive to overhead, FUNC and BB-5 might be better alternatives since they can reproduce all 13 tested bugs with mostly fewer than 5 replay attempts. Of course, their better capability in bug reproduction comes with the cost of increased overhead: they have decent performance for server (8–48% throughput degradation), but relatively high overhead (48–779%) (although still much smaller than previous work) for CPU-intensive SPLASH-2 benchmarks. FUNC has moderate overheads (18–84%) for desktop applications.

Above results indicate that PRES is quite feasible and can present multiple choices to developers based on their application need. *Note that the good bug reproduction results of low-overhead, but coarse-grained, sketching schemes such as SYS and SYNC are all due to our intelligent PI-Replayer that can leverage partial information from production runs and systematically explore the unrecorded non-deterministic space in an intelligent way (leveraging*

feedback from unsuccessful ones). For example, among the 12 bugs that SYNC and SYS reproduced, only 4 of them can be reproduced at their first attempts. All the other 8 bugs need more than one replay, relying on several rounds of feedback analysis to reproduce. Later in Section 7.2, we will compare the bug reproduction with and without feedback to further demonstrate the benefit.

BB-2 has better bug reproduction results in most cases but unfortunately incurs larger overheads than FUNC and BB-5. Interestingly, it is less efficient in reproducing a few bugs such as Barnes than FUNC and SYS. The reason is that its more detailed sketch information has caused the PRES monitor to detect more off-sketch replays and perform some unnecessary replay abort. Although a replay path may be different from production run (for example, it may take a few more iterations of waiting in a "while(flag)" loop), the bug could still be successfully reproduced if we continue the replay. Such case does not happen with BB because it contains more information to guide the execution to stick to the right path.

As Base records only uni-processor-related non-determinism, it cannot reproduce any tested bugs with 1000 tries, which matches previous results on concurrency testing [26] and also the reality faced by programmers. On the other hand, with all non-deterministic information, RW can reproduce all bugs at the first try, but its overhead is too large to bear (on average, 212X for non-servers, and up to 76.2% server throughput degradation).

Different types of applications have different characteristics (CPU bound vs I/O bound), and different data sharing and synchronization patterns. Therefore, the trade-offs of various sketching schemes are also different. For server applications, FUNC seems to present the best trade-offs. It only has 20% average throughput degradation, and can successfully reproduce the five evaluated server bugs within 1–24 replay attempts.

For desktop and scientific applications, SYNC provides the best trade-offs. Especially, those applications have less CPU idle time than server applications so that they have relatively higher recording overhead with the fine-grained sketching schemes like FUNC, BB-N, and BB. Moreover, as we said above, detailed sketch information may cause unnecessary replay abort especially at their customized synchronizations. Generally, the more threads are involved in a certain race condition (e.g., a customized barrier), the more replay attempts are required with fine-grained sketching schemes.

Different types of concurrency bugs may have different requirement on the amount of information needed for reproduction. Our results show that deadlocks are easier to reproduce than atomicity violation and order-violation bugs since the former in many cases are only related to synchronizations whereas the latter are usually related to shared memory accesses. SYNC reproduces 3 evaluated deadlocks on the 1st replay while adding only small (7%, 15%, and 33%) overhead during production runs. Of course, for deadlock that involves shared memory accesses in addition to synchronization operations, it may take multiple replay attempts for SYNC or SYS to reproduce.

7.2 The Effect of Feedback Generation

Our feedback mechanism has greatly improved the efficiency of bug reproduction and enabled the effectiveness of all PRES schemes, especially those coarse-grained ones such as SYNC and SYS. Table 6 uses three applications (Apache, PBZip2 and Barnes) as examples and compares the bug reproduction process between with and without feedback. As we can see, after a first unsuccessful replay attempt, **without feedback, no scheme, except BB-2 for Apache, can reproduce these three bugs within 1000 tries. In other words, without feedback, they are not much better than Base** (reasons explained in Section 5.2.1). With feedback, these bugs can be reproduced mostly within 10 tries. Similar results have also been observed for all other bugs and applications. These results demonstrate the significant benefit of feedback generation.

Applications		SYNC	SYS	FUNC	BB-5	BB-2	BB
Apache#1	w/	96th	28th	7th	8th	7th	1st
	w/o	NO	NO	NO	NO	754th	1st
PBZip2	w/	3rd	3rd	2nd	4th	3rd	3rd
	w/o	NO	NO	NO	NO	NO	NO
Barnes	w/	10th	10th	1st	74th	19th	1st
	w/o	NO	NO	1st	NO	NO	1st

Table 6: Benefit of feedback generation in bug reproduction — a comparison on # of replays needed to reproduce bugs. NO means that the bug was not successfully reproduced within 1000 tries.

Bug Id.	Base	SYNC	SYS	FUNC	BB-5	BB-2	BB
Apache#1	54390	1072	274	33	25	25	6
Apache#2	33176	190	97	12	11	8	3
MySQL#1	39983	2	2	2	2	1	0
Cherokee	133	86	58	16	36	7	3
Mozilla#1	36258	317	310	14	72	60	42
Mozilla#2	1067074	4	4	2	3	3	2
PBZip2	667	326	318	1	6	4	4
Transmission	255	240	172	6	6	6	4

Table 7: Effects of race filtering using sketch information. This table shows the number of *dynamic* benign data races to be explored with different sketch schemes. Base shows the number of dynamic benign races in the original execution. With different sketching schemes, PI-Replayer filters out those races whose *orders* at the original execution are determined by the global order of sketching points (e.g., function entry points).

7.3 The Effect of Race Filtering

When a replay fails, the feedback generator identifies dynamic data races in the unsuccessful replay, and picks one race pair to flip for the next replay. Since the execution order of some race pairs are already recorded or implied by the production-run sketch, PRES filters such order-determined races.

Table 7 shows the benefit of such filtering process. First, before any filtering (as shown with Base), applications can encounter many dynamic benign races during execution. Fortunately, since PRES records the global order of sketch points and thereby can determine the order for the significant number of such benign races, especially with fine-grained sketching schemes such as FUNC, BB-N, and BB. For example, with FUNC, only 1-33 benign races are left after filtering. Benefiting from this, the unrecorded non-deterministic space that PRES explores is significantly reduced to be practical.

7.4 Bug vs. Execution Reproduction

The PRES targets for bug reproduction instead of execution-path reproduction. However, since PRES strictly follows the sketch information from the production run, a successful bug-reproduction replay is still very similar to the original execution. As shown in Table 8, even with SYNC, a coarse-grained sketching scheme, the execution path of a successful bug-reproduction replay is only 0.11-0.19% different from those of the original execution.

We also modify PI-Replayer to claim success only after the execution path is reproduced. This is accomplished by recording only

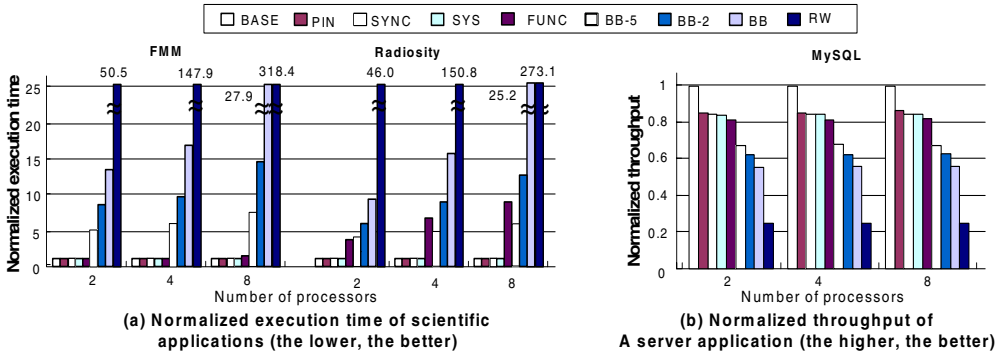

Figure 9: Scalability of PRES

Applications	SYNC	SYS	FUNC	BB-5	BB-2
Mozilla#2	0.11%	0.11%	0%	0%	0%
PBZip2	0.19%	0.19%	0%	0%	0%

Table 8: Percentage execution path difference between a successful bug-reproduction replay and the original execution. The execution path difference between two runs is calculated by using the editing distance between the basic-block traces of these two runs. A percentage is computed using the difference over the length of the basic block trace of the original execution.

Applications		SYNC	SYS	FUNC	BB-5	BB-2	BB
Mozilla#2	BR	3rd	3rd	3rd	4th	4th	3rd
	ER	16th	16th	4th	4rd	4th	3th
PBZip2	BR	3rd	3rd	2nd	4th	3rd	3rd
	ER	5th	5th	2nd	4th	5th	3rd

Table 9: The number of replays for bug reproduction (BR) vs. those for execution reproduction (ER)

local basic block traces from each thread during the original execution (with only slightly increased overhead since no synchronization is need to obtain global order). Then these traces are used to detect/abort off-path replay attempts.

Table 9 shows the number of replays required for each sketching scheme to reproduce the exact same execution as the original execution. While FUNC and BB do not require many additional replay attempts to reproduce the exact same execution path, SYS and SYNC require 1.6-5 times more attempts than when targeting for bug reproduction.

7.5 Scalability

We compare the overheads of various sketching schemes over different number of processors, 2, 4 and 8-cores in Figure 9. One factor would increase the relative overheads with the number of processors: more lock contention in obtaining the global order of recording points. On the other hand, the parallelization of log file writes (one log file per thread) can help reduce the relative overheads. For FMM and Radiosity, the first factor dominates, so the relative overheads all increase with the number of processors. As MySQL is more I/O bounded, the first factor has less effect on the performance, and therefore the relative overheads remain stable across all three configurations.

The relative overheads with SYNC, SYS, and FUNC are small across all three configurations. For example, in the case of FMM,

Applications		SYNC	SYS	FUNC	BB-5	BB-2	BB
Apache#1	CFF	96th	28th	7th	8th	7th	1st
	CBF	499th	116th	14th	10th	10th	1st
Mozilla#2	CFF	3rd	3rd	3rd	4th	4th	3rd
	CBF	5th	5th	3rd	3rd	4th	4th
PBZip2	CFF	3rd	3rd	2nd	4th	3rd	3rd
	CBF	17th	17th	13th	7th	8th	3rd

Table 10: A comparison between our Close-to-failure-first(CFF) ranking scheme and an alternative one, Close-to-beginning-first(CBF), for suspect selection

SYNC and SYS impose only 5.2% and 10.6% on 4-cores (in contrast to 636% overhead by SMP-Revirt [8]), respectively. In Radiosity, SYNC and SYS add 6.2% and 11.8% overhead on 2-cores (in contrast to 770% overhead by SMP-Revirt [8]).

7.6 The Effect of Suspect Selection Scheme

Our feedback mechanism explores the suspect data race set based on close-to-failure-first scheme. The table 10 compares our suspect selection scheme with an alternative one, referred as close-to-beginning-first. With the scheme, PRES picks the data race closest to the beginning of a program first. Note that it still uses depth-first-search, which means that during a replay by flipping a certain data race R, if more suspect races are observed on the new explored path, PRES picks one of them (based on close-to-beginning-first) for the next replay while still preserving the flipped order of R.

Since the coarse-grained schemes such as SYNC and SYS tend to have more unfiltered data races on a new explored path than other sketching schemes, the benefit of the close-to-failure-first scheme is more visible. From our observations, a bug has relatively short propagation period to a manifestation point, and therefore our close-to-beginning-first is better than the alternative.

7.7 Production-run Performance Details

The number of recording points The overhead differences among different schemes can be explained by their different recording granularity. In Table 11, SYNC and SYS conduct coarse-grained sketching, and their recording frequencies are much less intensive than other schemes. Consequently, they have smaller overhead than others.

Production-run log size In Table 11, SYNC and SYS have the smallest log sizes. Compared to hardware-based recording approaches, log size is much less critical in PRES because logs can be easily written to disks in software (in background). With the in-

Application		SYNC	SYS	FUNC	BB-5	BB-2	BB	RW
Apache	#RP(thousand/req)	1.03	1.71	190.10	128.52	317.17	768.92	2076.38
	Log size(KB/req)	2.00	3.34	371.30	251.01	619.47	1501.81	8110.88
MySQL	#RP(thousand/req)	0.65	0.65	27.00	52.21	129.54	258.83	861.50
	Log size(KB/req)	1.27	1.27	52.73	101.97	253.00	505.65	3365.22
Cherokee	#RP(thousand/req)	0.12	0.12	0.68	0.42	0.77	1.31	2.62
	Log size(KB/req)	0.95	0.95	5.42	3.32	6.17	10.44	41.85
OpenLDAP	#RP(thousand/req)	64.79	64.79	1784.64	2157.55	3764.65	6880.73	14838.75
	Log size(KB/req)	126.50	126.50	3485.63	4214.00	7352.86	13439.00	57963.86

Table 11: The number of recording points (#RP) and compressed log sizes. We use servers as examples. Other applications have similar results. We use KB/req instead of KB/s to measure log size so that the comparison is fair (since different schemes add different amounts of overhead).

Application		SYNC	SYS	FUNC	BB-5	BB-2	BB	RW
MySQL#1	replays	3rd	3rd	2nd	3rd	2nd	1st	1st
	time(s)	7.65	7.65	4.39	6.05	4.51	1.24	0.84
PBZip2	replays	3rd	3rd	2nd	4th	3rd	3rd	1st
	time(s)	73.25	74.47	23.48	37.56	71.52	78.66	7.14

Table 12: Bug reproduction time. Fine-grained sketching schemes take much longer to reproduce a bug, especially compared to RW, which represents previous software-only solutions. However, RW's good performance in bug reproduction comes with the prohibitive cost of high recording overhead during production runs, which is much more performance-critical.

creasing disk capacity and good sequential disk write performance, large log files are not a major concern as long as they are not terabytes per second. In addition, old log entries can be truncated especially with the support of checkpoint [36].

7.8 Bug Reproduction Time

In Table 12, we compare the bug reproduction time of each PRES scheme. As expected, fine-grained sketching schemes take much longer (up to 11 times) than RW, representing state-of-the-art software solutions for deterministic replay on multi-processors. PRES' bug reproduction time is affected by two factors: (1) the number of replay runs needed to reproduce the bug, and (2) the average length of each failed replay, since instrumentation and analysis are added to guide and monitor each replay. Therefore, the fewer the number of replay attempts or the faster PRES can abort an unsuccessful replay, the faster the bug reproduction time. For example, even though BB-5 needed more replay runs than BB-2 and BB in order to reproduce the bug in PBZip2, it takes less time because BB-2 and BB has more instrumentation overhead during each replay as they need to monitor every (or every other) basic block. Even though SYNC takes only 3 replays to reproduce the bug, its overheads are 9-10 times of RW's because it incurs more instrumentation and analysis overhead during each replay.

8. RELATED WORK

Most related work was discussed in Section 1, but here we focus on some that was not discussed in detail.

R2 [10] proposes an innovative method that replays an application by (1) recording the results of functions selected by programmers during production run, and (2) *returning the results* during replay from the log rather than *executing* the functions. The level of the selected functions plays an important role in overhead, because the lower the level is, the more information there is that needs to be recorded. In contrast, the higher the level is, the less detail there is that can be replayed for root-cause analysis, since those selected functions are not executed at all (i.e., their execution is "fast-forwarded"). The latter case may reproduce some failure symptoms without reproducing the bug, if the buggy code is "fast-forwarded" during replay [10].

The trade-off above is especially an issue for concurrency bugs on multi-processor machines, as acknowledged by the authors as a limitation of R2 (Section 7 in [10]). Concurrency bugs require *detailed* non-deterministic information (e.g., interleaving of shared memory accesses) to reproduce. Due to the overhead concern, R2 does not record such information. As a result, it is challenging for R2 to reproduce concurrency bugs. In contrast, reproducing concurrency bugs on multi-processors is our main objective.

Although our sketching schemes (especially FUNC and SYS) may sound similar to R2 [10], they are in fact very different: (1) Our sketches are used merely as guidelines by our PI-Replay for systematically *exploring* and *reconstructing* all missing *fine-grained*, non-deterministic information to reproduce the occurred bug at instruction-level granularity (the same as in previous deterministic replay work); whereas R2 replays only at a selected function granularity and skips the details below that level. (2) Our sketching schemes are used to record the global *order* of operations executed concurrently on multi-processors, not the return results or any side-effects. (3) Unlike R2, our sketching schemes are systematic, requiring no annotations from programmers. However, we can also use the information provided by R2 as a sketch to give to our PI-Replayer. Therefore, our PRES is complementary to R2.

The high level idea of PRES was also proposed by Stone in 1988 [33]. But it tries to demonstrate the idea by doing a customized (hard-coded) implementation for only one micro-benchmark, a shared queue, which is easy to analyze. In contrast, PRES provides a general framework that is not application-specific. In addition, that work passively uses whatever information available at production runs, whereas PRES actively records sketches during production runs and evaluates the trade-offs of various sketching schemes.

CHESS [20] is a novel approach that was recently proposed for exposing concurrency bugs. By systematically exploring the interleaving space, CHESS can expose some unknown concurrency bugs during testing. It works completely off-line at the developers' site. CHESS and PRES's replayer do share some similarities in that they both need to navigate a non-deterministic execution space, but CHESS has a very different objective from PRES. CHESS tries to find unexplored interleavings to test, while PRES uses execution sketches as guidelines and feedback from failed replays to get closer to reproducing a *target* concurrency bug. Since their ob-

jectives are different, their design, architecture, and issues are also different from each other in many apparent ways. It is the same when comparing PRES with other concurrency testing tools such as CTrigger [26].

Previous synchronization recording work Several systems [5, 20, 25, 28] have been proposed for recording and replaying the global order of synchronization operations. The SYNC scheme was inspired by these models, yet it is very different from them. First, the previous systems can help only uni-processor replay [5, 20, 28], or at best can replay only perfectly synchronized programs on multi-processors [25] (discussed in detail in Section 1.2). PRES, however, can reproduce concurrency bugs of general multi-threaded programs (without any modification) on multi-processors. PRES achieves this because, *once again, our sketches are only high-level guidelines for our PI-Replayer, which uses them to reconstruct missing non-deterministic information and reproduce the occurred concurrency bug on multi-processors*. For the same reason, unlike previous work [5, 20, 25, 28] , PRES does not require full knowledge of *all* synchronization operations, including even those defined by individual applications.

9. CONCLUSIONS AND FUTURE WORK

This paper has addressed one of the major challenges faced by the current multi-core era—-how to reproduce concurrency bugs on *multi-processors*. PRES trades replay-time efficiency *slightly* for significantly lower production-run recording overhead. Our evaluation using 11 real-world applications and 13 real world concurrency bugs has shown that PRES is effective and scalable. For example, SYNC and SYS reproduce all but one evaluated concurrency bug within a small number (most in < 10) of replay runs while imposing reasonably small overhead. These low-overhead but coarse-grained recording schemes are capable of reproducing concurrency bugs on multi-processors because our intelligent partial-information replayer can leverage feedback from unsuccessful replays to reconstruct the unrecorded non-deterministic information necessary for bug reproduction.

All work has limitations, and ours is no exception. (1) Although we wanted to evaluate more real world bugs, it is very time-consuming to find appropriate inputs and conditions for triggering a bug. We thought about injecting concurrency bugs into applications, but it is hard to guarantee their representativeness. (2) As shown in our experiments, some sketching methods such as SYNC and SYS still could not reproduce the bug for Radiosity after 1000 tries, which indicates the need to further enhance our PI-Replayer. (3) While PRES's overheads are significantly lower than those of previous methods, they may still be considered high for some performance-critical applications, so further optimizations will be useful. We plan to start by replacing Pin with a static instrumentation tool. We can also perform more aggressive code analysis to reduce the number of recording points. (4) We also need to extend PRES to support distributed applications. (5) It would be beneficial to implement PRES inside a virtual machine like SMP-Revirt [8] to make it easier to support replay on different physical machines.

While all our tested bugs can be successfully reproduced by PRES in our experiments and also theoretically it is feasible for PRES to exhaustively search the unrecorded non-determinism space to find a bug-producing combination, it is conceivable that in some cases it might be difficult for PRES to reproduce an occurred bug within an acceptable time limit. In particular, it might take a long time for PRES to produce a bug (1) that has a long error-propagation time before leading to a detectable failure, or (2) that requires *multiple* rare interleavings (i.e. race orders) to happen in certain ways to manifest. In case (1), PRES's closest-to-failure policy would take a long time to find the key race to flip. In case (2), the probability to reproduce such combination is too low unless some special corner-case based testing mechanism such as Ctrigger [26] is used during replay to easily force those corner-case interleavings (orders).

Finally, the five sketching mechanisms studied in this paper are just a starting point. We plan to utilize other sketching schemes. We also hope to inspire researchers to improve our schemes. Additionally, our main idea of relaxing the objective of "reproducing the bug on the first attempt" can be exploited by other approaches to enlarge their design space and lower their overhead.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback, the Opera groups for useful discussions and paper proofreading. We also thank our shepherd, Peter Chen, for his guidance in preparing the final version.

This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, and Intel gift grants.

11. REFERENCES

[1] Direct communication with the authors of SMP-Revirt, 2009.

[2] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP*, 1995.

[3] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, 2004.

[4] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, pages 319–328. ACM, 2008.

[5] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT*, 1998.

[6] J. Devietti, B. Lucia, M. Oskin, and L. Ceze. Dmp: Deterministic shared-memory multiprocessing. In *ASPLOS*, 2009.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.

[8] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[9] G. W. Dunlap. Execution replay for intrusion analysis (ph.d. thesis). http://www.eecs.umich.edu/ pmchen/papers/dunlap06.pdf.

[10] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.

[11] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.

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

[13] O. Laadan, R. A. Baratto, D. Phung, S. Potter, and J. Nieh. Dejaview: A personal virtual computer recorder. In *SOSP*, 2007.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4), 1987.

[15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[16] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, March 2008.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[18] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

[19] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: Abstractions and software-hardware interface for hardware-assisted deterministic multiprocessor replay. In *ASPLOS*, 2009.

[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

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

[22] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS*, 2006.

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

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

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient determistic multithreading in software. In *ASPLOS*, 2009.

[26] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[27] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, 1996.

[28] M. Ronsse and K. D. Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Automated and Algorithmic Debugging*, Nov 2000.

[29] S. Sarangi, B. Greskamp, and J. Torrellas. Cadre: Cycle-accurate deterministic replay for hardware debugging. In *DSN*, 2006.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[31] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[32] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.

[33] J. M. Stone. Debugging concurrent processes: a case study. In *SIGPLAN*, pages 145–153. ACM, 1988.

[34] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP*, 2007.

[35] VMware. (appendix c) using the integrated virtual debugger for visual studio. http://www.vmware.com/pdf/ws65_manual.pdf.

[36] VMware. Using the snapshot (vmware workstation 4). http://www.vmware.com/support/ws4/doc/preserve_snapshot_ws.html.

[37] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[39] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA/03*.

[40] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[41] M. Xu, M. D. Hill, and R. Bodík. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS*, 2006.