DoublePlay: Parallelizing Sequential Logging and Replay

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

Abstract

*Deterministic replay systems record and reproduce the execution of a hardware or software system. In contrast to replaying execution on uniprocessors, deterministic replay on multiprocessors is very challenging to implement efficiently because of the need to reproduce the order or values read by shared memory operations performed by multiple threads. In this paper, we present DoublePlay, a new way to efficiently guarantee replay on commodity multiprocessors. Our key insight is that one can use the simpler and faster mechanisms of single-processor record and replay, yet still achieve the scalability offered by multiple cores, by using an additional execution to parallelize the record and replay of an application. DoublePlay timeslices multiple threads on a single processor, then runs multiple time intervals (*epochs*) of the program concurrently on separate processors. This strategy, which we call* uniparallelism*, makes logging much easier because each epoch runs on a single processor (so threads in an epoch never simultaneously access the same memory) and different epochs operate on different copies of the memory. Thus, rather than logging the order of shared-memory accesses, we need only log the order in which threads in an epoch are timesliced on the processor. DoublePlay runs an additional execution of the program on multiple processors to generate checkpoints so that epochs run in parallel. We evaluate DoublePlay on a variety of client, server, and scientific parallel benchmarks; with spare cores, DoublePlay reduces logging overhead to an average of 15% with two worker threads and 28% with four threads.*

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Performance, Reliability

Keywords Deterministic Replay, Uniparallelism

1. Introduction

Deterministic replay systems record and reproduce the execution of a hardware or software system. The ability to faithfully reproduce an execution has proven useful in many areas, including debugging [15, 36, 40], fault tolerance [5], computer forensics [9], dynamic analysis [7, 26], and workload capture [22, 46].

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

Deterministic replay systems work by logging all nondeterministic events during a recording phase, then reproducing these events during a replay phase. On uniprocessors, nondeterministic events (e.g., interrupts and data from input devices) occur relatively infrequently, so logging and replaying them adds little overhead [46]. If there are multiple threads running on the uniprocessor, these can be replayed at low overhead by logging and reproducing the thread schedule [33]. On multiprocessors, however, shared-memory accesses add a high-frequency source of nondeterminism, and logging and replaying these accesses can drastically reduce performance.

Many ideas have been proposed to reduce the overhead of logging and replaying shared-memory, multithreaded programs on multiprocessors, but all fall short in some way. Some approaches require custom hardware [12, 20, 21, 45]. Other approaches cannot replay programs with data races [32] or are prohibitively slow for applications with a high degree of sharing [10]. Some recent approaches provide the ability to replay only while the recording is in progress [18] or sacrifice the guarantee of being able to replay the recorded execution without the possibility of a prohibitively long search [1, 30, 43, 47].

In this paper, we describe a new way to guarantee deterministic replay on commodity multiprocessors. Our method combines the simplicity and low recording overhead of logging a multithreaded program on a uniprocessor with the speed and scalability of executing that program on a multiprocessor.

Our key insight is that one can use the simpler and faster mechanisms of single-processor record and replay, yet still achieve the scalability offered by multiple cores, by using an additional execution to parallelize the record and replay of an application. Our goal is for the single-processor execution to be as fast as a traditional parallel execution, but to retain the ease-of-logging of single-processor multithreaded execution.

To accomplish this goal, we observe that there are (at least) two ways to run a multithreaded program on multiple processors, which we call *thread parallelism* and *epoch parallelism*. With thread parallelism, the threads of a multithreaded program run on multiple processors. While this traditional method of parallelization can achieve good scalability, it is expensive to log the shared-memory accesses from multiple threads running simultaneously. With epoch parallelism, multiple time intervals (*epochs*) of the program run concurrently. This style of parallelism has also been called Master/Slave Speculative Parallelism by Zilles [48] and Predictor/Executor by Süßkraut [39].

Uniparallel execution runs a thread-parallel and an epoch-parallel execution of a program concurrently. It further constrains the epoch-parallel execution so that all threads for a given epoch execute on a single processor. This strategy makes logging much easier because multiple threads in an epoch never simultaneously access the same memory, and because different epochs operate on different copies of the memory. Thus, rather than logging the order

of shared-memory accesses, we need only log the order in which threads in an epoch are timesliced on the processor.

To run epochs in parallel, we must generate checkpoints from which to start each of the epochs, and we must generate these checkpoints early enough to start future epochs before the earlier ones finish. Uniparallel execution generates these checkpoints speculatively by combining thread parallelism and online replay [18]. A thread-parallel execution runs ahead of the epoch-parallel execution and generates checkpoints from which to start future epochs. If these checkpoints diverge from the state produced by the epoch-parallel execution, we abort all epochs that started after the divergence, roll back, and restart from the last matching state. We reduce the chance of divergence by logging a subset of the non-deterministic events in the thread-parallel execution (e.g., the order of synchronization operations) and using this log to guide the execution of the epoch-parallel execution down a similar path.

This approach supports deterministic replay on commodity multiprocessors without needing to log shared-memory accesses. Replaying multithreaded programs on multiprocessors becomes as easy as replaying multithreaded programs on uniprocessors, while still preserving the speed and scalability of parallel execution. The main overhead of this approach is the use of more cores to run two instances of the program during logging. We believe the advent of many-core processors makes this a worthwhile tradeoff: the number of cores per computer is expected to grow exponentially, and scaling applications to use these extra cores is notoriously difficult.

To demonstrate these ideas, we implement a system called *DoublePlay* that can take advantage of spare cores to log multithreaded programs on multiprocessors at low overhead and guarantee being able to replay them deterministically. We evaluate the performance of DoublePlay on a variety of client, server, and scientific parallel benchmarks. We find that, with spare cores, DoublePlay increases run time by an average of 15% with two worker threads and 28% with four worker threads, and that DoublePlay can replay the run later without much additional overhead. On computers without spare cores, DoublePlay adds approximately 100% overhead for CPU-bound applications that can scale to 8 cores; this compares favorably with other software solutions that provide guaranteed deterministic replay.

Our paper is organized as follows. Section 2 describes other approaches to replaying multithreaded programs on multiprocessors. Section 3 discusses the principle of uniparallelism. Sections 4 and 5 describe the design and implementation of DoublePlay. Section 6 reports on how DoublePlay performs for a range of benchmarks, and Section 7 concludes.

2. Related work

Because of its wide array of uses, deterministic replay has been the subject of intense research by the hardware and software systems communities.

Many of the early replay systems focused on uniprocessor replay; e.g., IGOR [11], Hypervisor [5], Mach 3.0 Replay [33], DejaVu [6], ReVirt [9], and Flashback [36]. The designers of these systems observed that when executing a multithreaded program on a uniprocessor, there are many fewer thread switch events than accesses to shared memory. DoublePlay leverages this observation by logging and replaying epochs that each run the multithreaded program on a single processor, while taking advantage of multiple processors by starting multiple epochs in parallel.

Multiprocessor replay has been a particular area of focus in recent years because of the growing prevalence of multi-core processors. The main difficulty in replaying multithreaded programs on multiprocessors is logging and replaying shared memory accesses efficiently. One approach is to log the order of shared accesses [16], but this incurs high overhead. SMP-ReVirt [10] uses page protections to log only the order of conflicting accesses to memory pages, but this still incurs high overhead for some applications because of the cost of memory protection faults and false sharing. Another approach is to log and replay the values returned by load instructions [4, 23], but this also incurs high overhead.

One way to reduce the overhead of logging multiprocessors is to add hardware support. The most common strategy is to modify the cache coherence mechanism to log the information needed to infer the order of shared memory accesses [2, 12, 20, 21, 24, 42]. While these approaches are promising, we would like to support deterministic replay on commodity multiprocessors.

Another response to the high overhead of logging multiprocessors is to reduce the scope of programs that can be replayed. RecPlay [32] logs only explicit synchronization operations and so is unable to replay programs with unsynchronized accesses to shared memory (data races). DoublePlay also logs synchronization operations, but it uses these only as hints to guide the execution of the epoch-parallel run; DoublePlay guarantees deterministic replay for programs with and without data races by logging all nondeterminism in the epoch-parallel run.

Researchers have also tried to reduce logging overhead by relaxing the definition of deterministic replay. Instead of requiring that all instructions return the same data returned in the original run, these approaches provide slightly weaker guarantees, which still support the proposed uses of replay. PRES [30] and ODR [1] guarantee that all failures in the original run are also visible during replay, where failures are usually defined as the output of the program and program errors (e.g., assertions). Respec [18] guarantees that both the output and the ending state of the replayed execution match the logged execution. DoublePlay requires the same criteria as Respec for deterministic replay when evaluating whether the epoch-parallel execution matches the thread-parallel execution, since this guarantees that each checkpoint that starts a future epoch matches the ending state of the prior epoch.

Respec also distinguishes between online and offline replay [18]. Online replay refers to the ability to replay while the recording is in progress; offline replay refers to the ability to replay after the recording is complete. Respec provides only online replay. DoublePlay uses online replay, but only for the purpose of guiding the epoch-parallel execution. DoublePlay supports offline replay by logging and replaying the epoch-parallel execution.

Another way to reduce logging overhead is to shift work from the recording phase to the offline replay phase. To achieve this, recent research has investigated an alternate approach to deterministic replay based on *search* [1, 17, 30, 43, 47]. Rather than logging enough data to quickly replay an execution, these systems record a subset of information (e.g., synchronization operations or core dumps), then use that information to guide the search for an equivalent execution. The search space includes all possible orders of shared-memory accesses that are allowed by the logged information, so it grows exponentially with the number of racing accesses. With good heuristics, search-based replay can often find equivalent executions quickly, especially for programs with few racing accesses. However, because of the exponential search space, they may not be able to find an equivalent execution within a reasonable time frame. In addition, even if the search succeeds within a few tries, the execution of the program during search can be slowed by several orders of magnitude due to the need to log the detailed order of shared-memory accesses [30]. DoublePlay logs similar information as some of these systems (e.g., the SYS configuration in PRES [30]), so its recording should add a similar amount of overhead to the original application. One can view DoublePlay as using extra cores to search for an equivalent replay *during the original run*. However, while other systems risk not being able to later find an equivalent run quickly (or at all), DoublePlay verifies dur-

ing recording that the epoch-parallel execution matches the thread-parallel execution. While DoublePlay must execute intervals that contain races sequentially, this is unlikely to slow the program significantly because these intervals are likely to be a small fraction of the overall execution time.

Deterministic replay helps reproduce non-deterministic multiprocessor executions. An alternative approach is to ensure that all inter-thread communication is deterministic for a given input [3, 8, 13, 28]. This approach eliminates the need to log the order of shared-memory accesses. However, current solutions for deterministic execution only support programs that are free of data races [28], require language [13] or hardware [8] support, or only support programs with fork-join parallelism.

Finally, DoublePlay applies ideas from research on using speculation to run applications in parallel, such as thread-level speculation [29, 35, 37]. In particular, Master/Slave Speculative Parallelization [48], Speck [26], and Fast Track [14] also use a fast execution to start multiple slow executions in parallel. DoublePlay applies this idea in a new way by using a fast, thread-parallel execution on multiple processors to start multiple uniprocessor executions that execute threads sequentially on one processor. As both the thread-parallel and uniprocessor executions run the same program, DoublePlay could leverage the speculative control and data flow prediction scheme introduced in SlipStream [38] so the leading thread-parallel execution communicates values and branch outcomes to the trailing epoch-parallel execution, further reducing logging overhead.

In summary, DoublePlay distinguishes itself from prior software-only multiprocessor deterministic replay systems by adding little overhead to application execution time during recording while also guaranteeing that the recorded execution will be able to be replayed in the future in a reasonable amount of time. The cost of this guarantee is that DoublePlay must use additional cores to run two executions of the program during recording.

3. Uniparallelism

Many properties are difficult or slow to achieve on a multiprocessor but easy to achieve on a uniprocessor. For example, one can guarantee deterministic replay on a uniprocessor by recording external inputs and scheduling events such as interrupts, but on a multiprocessor, one must also record or infer the order or value of shared memory operations. Other properties that are easier to guarantee on a uniprocessor include sequential consistency and detecting or avoiding certain types of races. Importantly, the cost to provide these properties on a multiprocessor may be much higher than 2x for many benchmarks; techniques that work well on a uniprocessor may become infeasible when two threads execute concurrently on different cores.

Uniparallelism is a technique for achieving the benefits of executing on a uniprocessor, while still allowing application performance to scale with increasing processors. A uniparallel execution consists of both a thread-parallel and epoch-parallel execution of the same program. The epoch-parallel execution runs all threads of a given epoch on a single processor at a time; this enables the use of techniques that run only on a uniprocessor. Unlike a traditional thread-parallel execution that scales with the number of cores by running different threads on different cores, an epoch-parallel execution achieves scalability in a different way, namely by concurrently running different epochs (time slices) of the execution on multiple cores. Epoch-parallel execution thus requires the ability to predict future program states; such predictions are generated by running the second, thread-parallel execution concurrently. Consequently, for CPU-bound workloads, uniparallel execution requires twice the number of cores as a traditional execution, leading to an approximately 100% increase in CPU utilization and energy usage (assuming energy-proportional hardware).

Uniparallelism improves performance in two cases. First, if a workload is not CPU-bound or an application cannot scale to effectively use all cores of a multicore computer, the cost of uniparallelism can be much less than 100% (typically less than 20% in our experiments). Second, when the property being provided is more than twice as expensive on a multiprocessor as on a uniprocessor (as is often the case for deterministic replay), it is more efficient in terms of both time and energy to run the application twice with uniparallelism than to provide the property directly on the multiprocessor version of the application. For instance, even without spare cores, DoublePlay's energy and performance overhead is lower than that of all prior systems which guarantee deterministic replay on commodity multiprocessors.

4. Design overview

The goal of DoublePlay is to efficiently record the execution of a process or group of processes running on a multiprocessor such that the execution can later be deterministically replayed as many times as needed. DoublePlay is implemented inside the Linux operating system, and its boundary of record and replay is the process abstraction. The operating system itself is outside the boundary of record and replay, so DoublePlay records the results and order of system calls executed by the process and returns this data to the application during replay.

Figure 1 shows an overview of how DoublePlay records process execution. DoublePlay simultaneously runs two executions of the program being recorded. Each execution uses multiple cores. The execution on the left, which we refer to as the *thread-parallel execution*, runs multiple threads concurrently on the cores allocated to it. DoublePlay partitions the thread-parallel execution into time slices called *epochs*. At the beginning of each epoch, DoublePlay creates a copy-on-write checkpoint of the state of the thread-parallel execution.

The execution on the right, which we refer to as the *epoch-parallel execution*, runs each epoch on one of the cores allocated to it. All threads of a given epoch run on the same core, which simplifies the task of deterministically replaying the resulting execution. DoublePlay achieves good performance by running *different* epochs of the same process *concurrently*. As shown in Figure 1, both the thread-parallel execution and the epoch-parallel execution start running the first epoch simultaneously. However, the thread-parallel execution runs the epoch much faster because it uses more cores. When the thread-parallel execution reaches the start of the second epoch, DoublePlay checkpoints the process state and uses that state to start running the second epoch in the epoch-parallel execution. By the time the thread-parallel execution is running the fourth epoch, the epoch-parallel execution is able to fully utilize the four cores allocated to it. From this point on, the epoch-parallel execution can achieve speedup from parallelization roughly equivalent to that achieved by the thread-parallel execution.

During recording, DoublePlay saves three items that are sufficient to guarantee that the process execution can be replayed deterministically in the future. First, DoublePlay records the initial state of the process at the start of recording. Second, DoublePlay records the order and results of system calls, signals, and low-level synchronization operations in GNU `libc`. Finally, DoublePlay records the schedule of thread execution (i.e., when context switches between threads occur) of the epoch-parallel execution. Note that since each epoch is executed on a single core in the epoch-parallel execution, DoublePlay does not need to record the ordering or results of any shared memory operations performed by multiple threads; the three items above are sufficient to exactly recreate identical operations during replay.

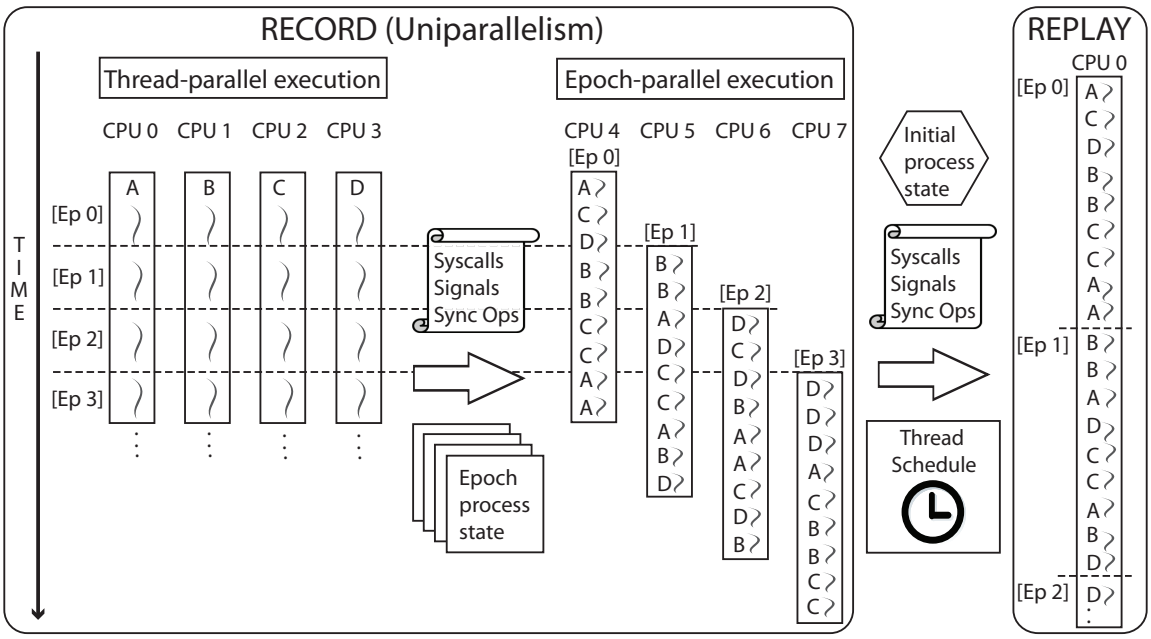

Figure 1. Overview of DoublePlay record and replay

The two executions can be viewed as follows. The epoch-parallel execution is the actual execution of the program that is being recorded. Because each epoch of the epoch-parallel execution runs on a single core, DoublePlay can use the simple and efficient mechanisms for uniprocessor deterministic replay to record and replay its execution. The thread-parallel execution allows the epoch-parallel execution to achieve good performance and scale with the number of cores. The thread-parallel execution provides a *hint* as to what the future state of the process execution will be at each epoch transition. As long as this hint is correct, the state of the process at the beginning of each epoch in the epoch-parallel execution will match the state of the process at the end of the previous epoch. This means that the epochs can be pieced together to form a single, natural execution of the process. This process is akin to splicing movie segments together to form a single coherent video.

But, what if the hint is incorrect? In such an instance, two epochs cannot be pieced together to form a single natural run; the logged run will contain unnatural transitions in program values at epoch boundaries akin to artifacts in a bad video splice. To detect such events, DoublePlay compares the process state (memory values and registers) of the thread-parallel and the epoch-parallel runs at each epoch transition. If the state of the epoch-parallel execution has diverged from the state of the thread-parallel execution, DoublePlay squashes the execution of all subsequent epochs for both executions. It restores the state of the thread-parallel execution to the checkpoint at the beginning of the epoch and restarts both executions from this point.

In Section 5.2.4, we discuss two different implementations for rolling back execution state. The first is a simpler design that rolls both executions back to the start of the epoch that diverged and restarts execution. The second is more complicated, but guarantees forward progress; it rolls both executions back to the state of the epoch-parallel execution at the divergence and restarts both executions from that state.

DoublePlay prevents the process from externalizing any output that depends on the execution of an epoch that may subsequently be rolled back. This means that output cannot be externalized until all prior epochs have been found to be free of divergence. Under the simpler design that rolls back to the start of the epoch, the current epoch must also be checked for divergence. Under the second design, output can be released as soon as the system call that generates that output is called by the epoch-parallel execution.

A divergence between the thread-parallel and epoch-parallel executions can slow performance substantially because it may lead to DoublePlay squashing the execution of several subsequent epochs. DoublePlay uses *online replay* [18] to reduce the frequency of such divergences and their resulting slowdown. During the thread-parallel execution, DoublePlay logs the ordering and results of all system calls and low-level synchronization operations in `libc`. When the epoch-parallel execution executes a system call or synchronization operation, DoublePlay returns the logged result instead of executing the operation. It also delays the thread execution to match the order of operations in the log. Further, DoublePlay logs signals and delivers them only on kernel entry or exit, making their effects deterministic. These actions are sufficient to ensure that the thread-parallel and epoch-parallel executions are identical (and, hence, do not diverge) for any epochs that are free of data races. The only situation in which the thread-parallel and epoch-parallel executions might diverge is when the program contains a data race and two threads execute unsynchronized, conflicting operations on a shared memory address.

At any subsequent time, DoublePlay can replay a recorded execution by (1) restoring the initial state of the recorded process, (2) replaying it on a single core using the logged system calls, signals, and synchronization operations, and (3) using the same schedule of thread execution that was used during the epoch-parallel execution.

Our current implementation only uses a single core during replay. However, DoublePlay could also use uniparallelism to speed up replay. In other words, DoublePlay could execute multiple threads of a replayed process on multiple cores and use the results to spawn concurrent replays of multiple epochs, each on a single core. As during recording, the executions could diverge, in which case, DoublePlay would squash future (incorrect) epochs and start again from the epoch that caused the divergence. This technique

would be useful, for example, to skip ahead to the next break point during debugging.

5. Implementation

The implementation of DoublePlay leverages our prior Respec [18] work to provide deterministic online replay. Respec ensures that two concurrent executions of the same process are *externally deterministic*, which we define to mean that the two executions execute the same system calls in the same order, and that the program state (values in the address space and registers) of the two processes are identical at the end of each epoch of execution.

Respec only ensures external determinism while two processes execute at the same time. It does not address deterministic replay of a process after the original execution completes; therefore, it cannot be used for offline debugging, intrusion analysis, and other activities that must take place after an execution finishes. DoublePlay builds on top of Respec to provide this capability.

We first provide background information about the design and implementation of Respec. We then describe how DoublePlay extends Respec to support offline deterministic replay.

5.1 Respec background

Respec is a system in the Linux kernel that runs two executions of a multithreaded process (or set of processes) concurrently. When Respec is invoked by a process, Respec duplicates the process's state via a *multi-threaded fork* that clones both its address space and the execution state of all concurrently executing threads. Respec then executes the cloned copy concurrently with the original execution.

The original and cloned executions have a producer-consumer relationship. When the original execution performs a system call, Respec writes the arguments and results of the call to a circular buffer. Respec logs the total order of system call entry and exit across all threads. During the cloned execution, Respec enforces the same total order by blocking threads when they enter or exit a system call until all prior entries in the circular buffer have been consumed. For most system calls, Respec does not re-execute the actual system call during the cloned execution; instead, it simply provides the results obtained by the original execution. Respec does re-execute system calls such as `clone` and `mmap` that change the process's address space to support subsequent loads and stores; it serializes these operations in both executions to ensure that they are performed deterministically. Respec also logs the value and timing of signals in this log, and delivers signals at the same point in both executions. To reduce log size, we modified Respec to assume that the underlying file system is copy-on-write [31, 34], so that it can delay logging large file reads until such files are actually modified. File system interfaces exist that let systems such as Respec create custom file versioning policies [41].

At user-level, Respec also writes to a circular buffer the low-level synchronization operations in `libc` performed by the original execution, such as lock acquisition, waits, and wakeups. In contrast to the system calls, Respec only records a happens-before partial order of synchronization operations and enforces this partial order during the cloned execution. Enforcing the recorded partial order for synchronization operations ensures that all shared memory accesses are ordered, provided the program is race free.

Combined, these mechanisms are sufficient to ensure that the original and cloned executions execute identically in the absence of data races. However, a benign or malign data race may cause the two executions to diverge. Respec divides the executions into semi-regular intervals called epochs. During each epoch, it compares the arguments passed to all system calls, and, at the end of each epoch, it verifies that the memory and register state of the original and cloned executions match. To reduce the cost of such comparisons, Respec only compares memory pages modified during the execution of the prior epoch.

Respec uses Speculator [25] to checkpoint the original execution at the beginning of each epoch. Each checkpoint is a copy-on-write fork of the process address space, so that only pages modified in an epoch are duplicated. Read-shared pages thus do not increase memory or cache overhead. If the divergence check succeeds, Respec commits the checkpoint. If the check fails, it rolls back the state of both executions to the checkpoint and tries the execution again. During an epoch, the recording process is prevented from committing any external output (e.g., writing to the console or network). Instead, its outputs are buffered in the kernel. Outputs buffered during an epoch are only externalized after both processes have finished executing the epoch and the divergence check succeeds. They are discarded on a rollback. Thus, epochs that are later rolled back are not visible to any external observer.

To summarize, the guarantees provided by Respec are: (1) the original and cloned executions produce the same external outputs (more precisely, the same system calls) in the same order, and (2) the memory and register state of the two executions are identical at epoch boundaries.

5.2 Supporting offline replay

DoublePlay uses Respec during recording to coordinate the thread-parallel and epoch-parallel executions. From the point of view of Respec, the thread-parallel execution is the original execution and the epoch-parallel execution is the cloned execution. DoublePlay makes several enhancements to the basic Respec infrastructure in order to support offline replay, which we describe in the following sections.

5.2.1 Enabling concurrent epoch execution

DoublePlay needs to run multiple epochs concurrently, while Respec runs only a single epoch at a time. DoublePlay therefore makes multiple copies of the thread-parallel execution by calling the multi-threaded fork primitive before starting the execution of each individual epoch. This primitive creates a new process whose state is identical to that of the thread-parallel execution at that point in its execution. Each time a new process is created, DoublePlay does not let it begin execution, but instead places it in an *epoch queue* ordered by process creation time.

The DoublePlay scheduler is responsible for deciding when and where each process will run. Currently, the scheduler uses a simple policy that reserves half of the available cores for the thread-parallel execution and half for the epoch-parallel execution. It uses the Linux *sched_setaffinity* system call to constrain process execution to specific cores. As long as cores remain available, the scheduler pulls the next process from the epoch queue, allocates a core to it, constrains it to execute on only that core, and wakes up the process. When the process completes executing the epoch, it informs the scheduler that the core is now free, and the scheduler allocates the core to the next process in the epoch queue.

Even though DoublePlay starts execution of the epochs in the epoch-parallel execution in sequential order, there is no guarantee that epoch execution will *finish* in order, since some epochs are much shorter than others. DoublePlay uses an adaptive algorithm to vary epoch lengths. It sets the epoch length to 50 ms after a rollback. Each epoch without a rollback increases the epoch length by 50 ms up to a maximum of one second. Further, by leveraging output-triggered commits [27], DoublePlay ends an epoch immediately if a system call requires synchronous external output and no later than 50 ms after asynchronous external output. Network applications have much external output, so they have many short epochs even with few rollbacks.

Even though epochs may finish out of order, DoublePlay ensures that they commit or roll back in sequential order. When a epoch completes execution, DoublePlay performs a divergence check by comparing its memory and register state to that of the checkpoint associated with the next epoch. Once this check passes, DoublePlay allows the process to exit and allocates its processor to another epoch. If all prior epochs have been committed, DoublePlay also commits the epoch and discards the checkpoint for that epoch. Otherwise, it simply marks the epoch as completed. After all prior epochs commit, DoublePlay will commit that epoch.

In its strictest form of verification, DoublePlay considers a divergence check to fail if (1) a thread in the epoch-parallel execution calls a different system call from the one called by the corresponding thread in the thread-parallel execution, (2) the two threads call different `libc` synchronization operations, (3) the two threads call the same system call or synchronization operation, but with different arguments, or (4) the registers or memory state of the two executions differ at the end of the epoch. In Section 5.2.5, we describe how we loosen these restrictions slightly to reduce unnecessary rollbacks.

When a divergence check fails, DoublePlay terminates all threads executing the current epoch and any future epochs in the epoch-parallel execution, as well as all threads of the thread-parallel execution, by sending them kill signals. However, epochs started prior to the one that failed the divergence check may still be executing for the epoch-parallel execution. DoublePlay allows these epochs to finish and complete their divergence checks. If these checks succeed, DoublePlay restarts the thread-parallel execution from the failed epoch. If a check for one of the prior epochs fails, it restarts execution from the earliest epoch that failed.

5.2.2 Replaying thread schedules

DoublePlay guarantees deterministic offline replay by executing each epoch on a single core using the same thread schedule that was used during recording by the epoch-parallel execution. There are two basic strategies for providing this property. One strategy is to use the same *deterministic scheduler* during epoch-parallel execution and offline replay. The second strategy is to *log the scheduling decisions* made during epoch-parallel execution and replay those decisions deterministically during offline replay.

To implement these strategies, we added a custom scheduler layer that chooses exactly one thread at a time to be run by the Linux scheduler and blocks all other threads on a wait queue. When the DoublePlay scheduler decides to execute a new thread, it blocks the previously-executing thread on the wait queue and unblocks only the thread that it chooses to run. It would also have been possible to implement these strategies by modifying the Linux scheduler, but this would have required instrumenting all scheduling decisions made in Linux.

To implement the first strategy (the deterministic scheduler), DoublePlay assigns a strict priority to each thread based on the order in which the threads are created. DoublePlay always chooses to run the highest-priority thread eligible to run that would preserve the total ordering of system calls and the partial ordering of synchronization operations.

To preserve system call ordering, the thread-parallel execution assigns a sequence number to every system call entry or exit when it is added to the circular buffer. A thread in the epoch-parallel execution only consumes an entry if it has a sequence number one greater than the entry last consumed. Thus, a high-priority thread whose next entry is several sequence numbers in the future must block until low-priority threads consume the intervening entries. Similarly, DoublePlay uses multiple sequence numbers to represent the partial order of user-level synchronization operations. Specifically, the address of each lock or futex accessed in a synchronization operation is hashed to one of 512 separate counters, each of which represents a separate sequence number. Note that once a low-priority thread consumes an entry in either buffer, a higher-priority thread may become eligible to run. In this instance, DoublePlay immediately blocks the low-priority thread and unblocks the high-priority thread. For any given set of system calls and synchronization operations, this algorithm produces a deterministic schedule. That is, context switches always occur at the same point in each thread's execution.

This first strategy is relatively easy to implement and requires no additional logging. However, it is difficult to allow preemptions in this strategy, because doing so would require inserting preemptions deterministically [28]. Allowing preemptions has two benefits: it allows uniparallel to reproduce any bug that manifests on sequentially-consistent hardware, and it maintains liveness in the presence of spin locks.

To allow preemptions, we implemented a second strategy for replaying thread schedules deterministically, which is to log the preemptions that occur during epoch-parallel execution and replay those decisions deterministically during offline replay. In order to deterministically reproduce preemptions, we record the instruction pointer and branch count of a thread when it is preempted during epoch-parallel execution. The branch count is necessary because the instruction could be inside a loop, and we must replay the preemption on the correct iteration [19]. These branch counts are maintained per thread and obtained using hardware performance monitoring counters configured to count branches executed in user-space [5]. We compensate for return from interrupt (`iret`) branches, which would otherwise cause interrupts to perturb the branch count non-deterministically. After recording the instruction pointer and branch count, we unblock the next thread. In offline replay, we preempt a thread when it reaches the recorded instruction pointer and branch count and allow the next thread to run.

As with all deterministic replay systems, DoublePlay perturbs the execution that it is logging. When using deterministic replay for debugging, such perturbations may change the likelihood of encountering particular bugs. Epoch parallelism avoids the perturbation due to instrumenting shared memory accesses, but adds perturbation due to timeslicing a multithreaded program onto a uniprocessor. The degree to which timeslicing changes the likelihood that a particular bug will manifest depends on the particular scheduling algorithm being used (e.g., certain race conditions may occur infrequently with coarse-grained timeslicing).

5.2.3 Offline replay

To support offline replay, DoublePlay records the system calls and synchronization operations executed during an epoch in a set of log files (for simplicity, DoublePlay uses a separate log for each thread). After committing each epoch, DoublePlay marks the entries belonging to that epoch as eligible to be written to disk. It then writes the marked records out asynchronously while other epochs are executing. Note that DoublePlay only has to record the results of synchronization operations and system calls, since the arguments to those calls will be deterministically reproduced by any offline replay process. This reduces log size considerably for system calls such as `write`. Signals are logged with the system calls after which they are delivered.

If a rollback occurs, DoublePlay deallocates any records in the circular buffer that occurred after the point in the thread-parallel execution to which it is rolling back (these records cannot have been written to disk since the epoch has not yet committed). It also ensures that all entries that precede the rollback point are written to disk before restoring the checkpoint. The checkpoint includes the sequence numbers at the point in execution where the checkpoint was taken, so subsequent entries will have the correct

sequence number. Thus, there is no indication of the divergence or the rollback in the logs on disk.

DoublePlay saves the initial state of the process when recording began. To perform an offline replay, it starts from this initial copy. DoublePlay currently runs the offline replay on a single processor and uses the scheduling algorithm described in Section 5.2.2 to constrain the order of thread execution for the offline replay process. When an offline replay thread executes a system call or synchronization operation, DoublePlay returns the results recorded in its log file. The only system calls that DoublePlay actually executes are ones that modify the process address space, such as `mmap` and `clone`. DoublePlay delivers recorded signals at the same point in process execution that they were delivered during recording.

As mentioned in Section 4, DoublePlay could potentially use uniparallel execution to accelerate replay in the same manner that it accelerates recording. With this optimization, we would expect offline replay execution time to be no worse than recording time.

5.2.4 Forward recovery

We implemented two different rollback strategies in DoublePlay. Initially, we decided to roll both executions back to the checkpoint at the beginning of the epoch that failed the divergence check. Both executions would restart from this point. If the divergence check again failed, we would roll back and try again. However, we saw some executions in which a given epoch would roll back several times in a row before the divergence check succeeded, presumably because it contained one or more frequently-diverging data races. Frequent rollbacks imposed a substantial performance overhead for some applications; in the worst case, a program with many frequent races could fail to make forward progress.

After consideration, we realized that this strategy reflected the incorrect view that the thread-parallel execution was the run being recorded. In fact, the epoch-parallel execution is the "real" execution being recorded — it is after all the one that is being executed on a single core with a known thread schedule. The epoch-parallel execution is also a perfectly legal execution that could have occurred on the thread-parallel execution with a particular set of relative speeds among the processors (since we verify the ending state of the prior epoch matches the starting state of the next epoch). The thread-parallel execution exists merely as a means for generating hints about future process state so that multiple epochs can be executed in parallel.

Once we viewed the epoch-parallel execution as the one being recorded, it was clear that the state of its execution at the time the divergence check fails is a valid execution state that can be deterministically reproduced offline. Therefore, DoublePlay can use the epoch-parallel process state at the time the divergence check fails as a checkpoint from which to restart execution. We call this process *forward recovery*.

A complication arises because the kernel state is associated only with the thread-parallel execution (because it is the process that actually executes all system calls), while the correct process state is associated with the epoch-parallel execution. DoublePlay detects when the thread-parallel and epoch-parallel executions diverge by comparing the order and arguments of system calls, and the memory and registers after each epoch. At the point of divergence, the (logical) kernel states of the two executions are guaranteed to be identical because the executions have issued the same sequence of system calls up to that point. Because the logical kernel states are identical, it is correct to merge the kernel state of the thread-parallel execution with the memory and register state of the epoch-parallel execution.

DoublePlay logs an undo operation for each system call that modifies kernel state, so that forward (and regular) recovery can roll back kernel state by applying the undo operations. It can thus roll back the thread-parallel execution's kernel state to the system call at which a divergence was detected. It then makes the contents of the address space of the thread-parallel execution equal to the contents of the address space of the epoch-parallel execution at the point where the divergence check failed.

For instance, consider a process with a data race that causes the memory states of the two executions to differ at the end of an epoch. DoublePlay's divergence check already compares the address space of the epoch-parallel execution with a checkpoint of the thread-parallel execution. If any bytes differ, DoublePlay simply copies the corresponding bytes from the epoch-parallel address space to the checkpoint's. DoublePlay then restores the checkpoint to restart the thread-parallel execution; new epoch-parallel executions will be spawned as it proceeds.

Forward recovery guarantees that DoublePlay makes forward progress, even when a divergence check fails. All work done by the epoch-parallel execution up until the divergence check, including at least one data race (the one that triggered the divergence), will be preserved. In the worst case, every epoch may contain frequent data races, and divergence checks might always fail. However, even in this case, DoublePlay should be able to approach the speed of uniprocessor replay since a single core can always make progress. In the expected scenario where data races are relatively infrequent, DoublePlay runs much faster.

5.2.5 Looser divergence checks

In our initial design for DoublePlay, we made the divergence check very strict because we wanted to detect a divergence as soon as possible in order to minimize the amount of work thrown away on a rollback. However, once we implemented forward recovery, we decided that strict divergence checks might no longer be best. As long as the epoch-parallel execution can continue its execution, any work that it completes will be preserved after a rollback.

Even with forward recovery, it is still necessary to check memory and register state at each epoch boundary. If the process state at the end of an epoch of the epoch-parallel execution does not match the state from which the next epoch execution starts, the epoch-parallel executions of all subsequent epochs are invalid and must be squashed. However, strict divergence checks are not necessarily required *within* an epoch.

As long as the external output of the epoch-parallel execution continues to match the external output of the thread-parallel execution, then the system calls and synchronization operations performed by the two executions can be allowed to diverge slightly. On the other hand, if one of these operations can affect state external to the process, then the state changes cannot be made visible to an external entity and all subsequent results from system calls and synchronization operations must be consistent with the divergence.

Based on this observation, we modified DoublePlay to support three slightly looser forms of divergence checks within an epoch. First, if a thread's circular buffer contains a system call with no effect outside that thread, but the epoch-parallel execution omits that system call, we allow it to skip over the extra record in the buffer. Examples of such system calls are `getpid` and `nanosleep`.

Second, if a thread executes a system call that produces output for an external device such as the screen or network, the epoch-parallel execution is allowed to execute the same system call with different output. Because DoublePlay buffers such output in the kernel until after both executions have completed the system call, no external entity observes this output prior to the execution of the call by the epoch-parallel execution. Because that execution is logically the "real" execution, we simply release its output, rather than the output from the thread-parallel execution, to external observers. The observers therefore see the same output during recording and offline replay. The output produced by the thread-

parallel execution can be viewed as an incorrect hint that is later corrected.

Third, if a thread executes a set of self-canceling synchronization operations or system calls during the thread-parallel execution, but omits them in the epoch-parallel execution, then all members of the set can be skipped. By self-canceling, we mean two or more operations that when executed together have no effect on state external to the thread. For example, lock and unlock operations for the same low-level lock are a self-canceling pair. If the thread-parallel execution performed both operations, then the epoch-parallel execution can achieve the same effect by performing neither.

Looser divergence checks have two benefits. First, the epoch parallel execution can run longer before a rollback is needed. Second, a rollback can sometimes be avoided all together when the divergence in process state is transitory. For example, one application we tested (pbzip2) has a benign race in which one thread spins waiting for another to set a value. Since the thread calls `nanosleep` periodically, the spin loop executes different numbers of iterations in different executions. With strict divergence checks, such behavior leads to rollbacks. However, with looser divergence checks, the epoch-parallel execution continues past this loop. While different calls to `nanosleep` leave temporary differences on the thread's stack, these differences are soon overwritten by subsequent system calls. Thus, unless the epoch boundary happens immediately after the spin loop, the divergence in program state heals and no rollback is needed.

5.2.6 Reverse scheduling

We implemented one final feature as an option to the deterministic scheduling algorithm, which we refer to as reverse scheduling. The idea behind reverse scheduling is to try to avoid rollbacks when a divergence check fails by finding a different schedule whose execution causes the check to pass. Thus, when a check fails, DoublePlay runs another epoch-parallel execution of the failed epoch on the processor allocated for that epoch, but it uses the opposite scheduling priorities from the ones it used in the failed execution. That is, the last thread created has the highest, not the lowest, priority. The idea is that if there is a single data race in the epoch, then either the normal or reverse schedule is very likely to reproduce the same program state produced by the thread-parallel execution. If the second execution passes the divergence check, the epoch is committed and no rollback occurs. The reverse thread schedule is recorded in the DoublePlay log so that it is also used during replay.

Our results with reverse scheduling were mixed. We found that this feature did in fact eliminate many rollbacks. However, for the applications we tested, fewer rollbacks did not lead to improved performance. One reason is that the forward recovery and loose replay optimizations already eliminated many rollbacks entirely and reduced the performance impact of the remaining ones. Another reason is that reverse scheduling may use CPU time for the second execution yet not eliminate the need for rollback. Thus, while reverse scheduling may prove useful for applications we have not yet tested, DoublePlay currently does not use this feature.

6. Evaluation

Our evaluation answers the following questions:

- What is the overhead of DoublePlay record and replay for common applications and benchmarks?
- How often do applications roll back, and what is the effect of rollback on replay time?
- Do our optimizations, namely forward recovery and loose replay, reduce overhead for applications with data races?

6.1 Methodology

We used two different 8-core computers to parallelize our evaluation. The first has a 2 GHz 8-core Xeon processor with 3 GB of RAM, while the second has a 2.66 GHz 8-core Xeon with 4 GB of RAM. Both computers run CentOS Linux version 5.3. The kernel is a stock Linux 2.6.26 kernel, modified to include DoublePlay. We also modified the GNU glibc library version 2.5.1 to support DoublePlay. We use our first strategy for replaying thread schedules, i.e. the deterministic scheduler that runs threads according to a strict priority.

We evaluated DoublePlay with 9 benchmarks: five parallel applications (pfscan, pbzip2, aget, the Apache web server, and the mysql database server) and 4 SPLASH-2 [44] benchmarks (fft, radix, ocean, and water). We report three values for each experiment: the original execution time of the application running on a stock system, the execution time during DoublePlay recording, and the execution time for DoublePlay offline replay. The record time measures the time for both the thread-parallel and epoch-parallel executions to finish.

DoublePlay periodically writes the kernel and user-level replay logs to disk so they can be used for offline replay. We report the size of the log and include the time taken to write the log in the recording cost.

We evaluate pbzip2 compressing a 311 MB log file in parallel. We use pfscan to search in parallel for a string in a directory with 952 MB of log files. We extended the benchmark to perform 100 iterations of the search so that we could measure the overhead of deterministic replay over a longer run while ensuring that data is in the file cache (otherwise, our benchmark would be disk-bound). Aget is a bandwidth-stealing application that takes advantage of I/O parallelism by opening multiple connections to a remote server. We used aget to retrieve a 21 MB file over a local network from a server configured to limit each connection to a maximum download bit rate of 1MB/sec. We tested Apache using ab (Apache Bench) to simultaneously send 100 requests from eight concurrent clients over a local network. We evaluate mysql using sysbench version 0.4.12. This benchmark generates 3000 total database queries on a 9 GB myISAM database; 2000 queries are read-only, 600 are updates, and 400 are other types such as table lock and unlock. Since mysql uses a separate worker thread to handle each client, we vary the number of concurrent clients depending on the number of worker threads we are evaluating in each trial.

For each benchmark, we vary the number of worker threads from one to eight for the original execution. DoublePlay uses two cores per worker thread, so we measure its performance with up to four worker threads. Many benchmarks have additional control threads which do little work during the execution; we do not count these in the number of threads. Pbzip2 uses two additional threads: one to read file data and one to write the output; these threads are also not counted in the number of threads shown. Unless otherwise mentioned, all results are the mean of five trials.

6.2 Record and replay performance

Table 1 shows the overall performance results for DoublePlay. The first three columns show the application or benchmark executed, the number of worker threads used, and the execution time of the application without DoublePlay. The next seven columns give statistics about DoublePlay execution: the number of user-level synchronization operations logged, system calls logged, epochs executed, memory pages compared, size of the DoublePlay logs written to disk, the average number of rollbacks that occurred per execution, and the time to record an execution. The next column shows the overhead added by DoublePlay during recording, compared to two configurations of the original application. The first overhead is relative to the original application with the same number of appli-

app	worker threads	original time (s) & stdev.	synch. ops.	system calls	epochs	pages compared	log size (MB)	average rollbacks per run	record time (s) & stdev.	recording overhead	offline time (s) & stdev.
pfscan	1	193.93 (0.11)									
	2	100.99 (0.89)	23302	7528	101	1325	1.60	0	108.38 (0.87)	7% / 105%	193.73 (0.26)
	3	69.01 (0.26)	22250	7531	60	868	1.55	0	78.91 (1.01)	14% / 112%	190.23 (2.88)
	4	52.97 (0.18)	21575	7534	50	760	1.52	0	59.20 (0.92)	12% / 102%	188.19 (1.16)
	6	37.18 (0.04)									
	8	29.26 (0.03)									
pbzip2	1	91.65 (0.06)									
	2	46.08 (0.03)	26182	5481	46	573243	1.30	0	50.42 (0.41)	9% / 111%	92.88 (0.11)
	3	31.45 (0.77)	26025	5207	31	574131	1.28	0	35.92 (0.09)	14% / 119%	92.94 (0.29)
	4	23.94 (0.38)	26393	5066	23	562705	1.29	0	29.38 (0.43)	23% / 128%	92.85 (0.10)
	6	16.38 (0.07)									
	8	12.90 (0.03)									
aget	1	21.19 (0.04)									
	2	10.73 (0.02)	25243	33495	22	267	27.41	0.4	10.80 (0.06)	1% / 99%	0.28 (0.01)
	3	7.22 (0.01)	21564	29022	20	263	26.54	0.2	7.31 (0.08)	1% / 102%	0.26 (0.01)
	4	5.42 (0.01)	19618	27372	20	277	26.13	0.2	5.54 (0.15)	2% / 104%	0.25 (0.02)
	6	3.62 (0.01)									
	8	2.71 (0.00)									
apache	1	44.36 (0.13)									
	2	43.59 (0.27)	3756	3944	393	5264	0.14	0.1	43.95 (0.33)	1% / 10%	0.04 (0.00)
	3	41.67 (0.35)	3682	3923	386	5285	0.14	1.0	42.74 (0.59)	3% / 11%	0.04 (0.00)
	4	40.13 (0.27)	3636	3904	389	5383	0.14	1.7	40.75 (0.65)	2% / 9%	0.04 (0.00)
	6	38.39 (0.41)									
	8	37.35 (0.61)									
mysql	1	29.97 (0.08)									
	2	29.25 (0.09)	195903	46691	3035	195903	13.86	0	34.89 (0.23)	19% / 19%	0.53 (0.00)
	3	29.28 (0.08)	199952	46792	3052	199952	14.07	0	34.98 (0.21)	19% / 19%	0.54 (0.00)
	4	29.20 (0.08)	200598	46951	3069	200598	14.10	0.2	34.25 (1.39)	17% / 17%	0.55 (0.00)
	6	29.33 (0.12)									
	8	29.35 (0.14)									
fft	1	117.88 (0.19)									
	2	58.12 (0.06)	15689	3011	67	547619	0.90	0	68.72 (0.25)	18% / 106%	115.65 (0.11)
	4	33.33 (1.08)	32242	3041	41	333837	1.77	0	43.47 (0.17)	30% / 131%	106.61 (0.15)
	8	18.79 (0.15)									
radix	1	177.84 (0.96)									
	2	89.10 (0.39)	4571	471	41	1295817	0.22	0	96.88 (0.15)	9% / 114%	177.58 (0.23)
	4	45.28 (0.28)	11140	607	41	1313664	0.53	0	53.43 (0.23)	18% / 127%	177.73 (0.24)
	8	23.50 (0.03)									
ocean	1	56.67 (0.03)									
	2	28.19 (0.67)	108808	149	32	914104	4.88	0	46.09 (0.06)	63% / 222%	56.45 (0.03)
	4	14.31 (1.35)	218788	222	27	745697	10.44	0	31.75 (0.26)	121% / 278%	53.24 (0.25)
	8	8.39 (0.09)									
water	1	154.57 (1.97)									
	2	81.70 (1.95)	5376008	21404	88	275484	207.33	0	89.00 (3.78)	9% / 106%	160.57 (3.63)
	3	57.78 (1.25)	6756393	21741	65	279701	260.64	0	63.10 (1.45)	9% / 89%	160.60 (2.13)
	4	43.15 (0.03)	8131526	21537	54	334339	313.81	0	56.32 (1.96)	31% / 92%	162.99 (1.05)
	6	33.39 (0.52)									
	8	29.33 (0.22)									

Results are the mean of five trials. Values in parentheses show standard deviations. Note that DoublePlay uses more cores than the original execution during recording since it executes two copies of the application. The overhead column shows the overhead of DoublePlay with respect to two configurations of the original application. The first overhead is relative to the original application with the same number of application threads; this shows the overhead of DoublePlay when it can take advantage of unused cores. The second overhead is relative to the original application with twice as many application threads; this shows the overhead of DoublePlay relative to an application configured to use the same number of cores as DoublePlay.

Table 1. DoublePlay performance

cation threads; this shows the overhead of DoublePlay when it can take advantage of unused cores. The second overhead is relative to the original application with twice as many application threads; this shows the overhead of DoublePlay relative to an application configured to use the same number of cores as DoublePlay. The last column shows the offline replay execution time.

app	threads	rollbacks w/o opt.	rollbacks with opt.	execution time(s) & stdev. w/o opt.	execution time(s) & stdev. with opt.	relative reduction in exec. time
pbzip2	2	2.2	0	53.32 (2.61)	50.42 (0.41)	6%
	3	1.2	0	38.54 (3.77)	35.92 (0.09)	8%
	4	0.8	0	32.59 (3.04)	29.38 (0.43)	13%
aget	2	0.4	0.4	10.80 (0.06)	10.84 (0.12)	0%
	3	0.2	0.2	7.31 (0.08)	7.30 (0.02)	0%
	4	0.2	0.2	5.54 (0.15)	5.57 (0.03)	0%
apache	2	0.1	0.1	43.95 (0.33)	43.81 (0.31)	0%
	3	1.0	1.0	42.74 (0.59)	41.78 (0.48)	2%
	4	1.7	1.7	40.75 (0.65)	41.09 (1.28)	-1%
mysql	4	0.2	0.2	34.25 (1.39)	34.25 (1.39)	0%

Table 2. Benefit of forward recovery and loose replay.

The availability of unused cores significantly impacts the overhead added by DoublePlay during recording. If there are sufficient unused cores, DoublePlay adds little overhead to the recorded application execution time. On average, 2 worker threads add about 15% overhead to the application run time. The overhead gradually increases to 28% with 4 threads. For the five real applications, the maximum overhead of any benchmark is 23% (for pbzip2 with 4 worker threads) — the average overhead is only 7% with 2 worker threads and 11% with 4 worker threads. Apache and aget are limited by the speed of our local network; mysql is limited by disk I/O, and the remaining benchmarks are CPU bound. DoublePlay shows more overhead for the SPLASH-2 benchmarks, generally because they perform many more synchronization operations per second and dirty memory pages more rapidly, which increases the number of pages compared. As shown in the Respec paper [18], Ocean is a challenging benchmark as most of its overhead derives from sharing the limited memory bandwidth and processor caches between the thread-parallel and epoch-parallel executions. As expected, Ocean incurs approximately the same overhead with DoublePlay as it incurs with Respec.

If all cores can be productively used by the application, then DoublePlay incurs much higher overhead because it executes the application twice during recording, and this uses cores that could have been used by the application. When compared to an application configured to use the same number of cores as DoublePlay, DoublePlay adds approximately 100% overhead to CPU-bound applications that can scale to eight cores (this does not affect Apache, which is network bound, or mysql, which is disk bound). For comparison, iDNA adds an average of 1100% overhead [4], and SMP-ReVirt adds 10-600% overhead [10]. SMP-ReVirt does not incur DoublePlay's overhead of executing the application twice, but SMP-ReVirt does incur high overhead for applications that share data frequently (including false sharing due to tracking ownership at page granularity) because of the cost of memory protection faults. DoublePlay also scales well up to 8 cores (e.g., its average overhead across all benchmarks without spare cores is 99% with 2 threads and 110% with 4 threads), while SMP-ReVirt does not scale well for some applications even up to 4 cores.

Thus, DoublePlay is well suited for three settings that require deterministic replay (e.g., for forensic or auditing purposes): (1) applications which cannot scale effectively to use all cores on the machine, (2) sites that are willing to dedicate extra cores to provide deterministic replay, and (3) applications that share data frequently between cores (for which DoublePlay is the lowest overhead solution). In particular, we believe that many machines will have unused cores that DoublePlay can take advantage of because the number of cores per computer is expected to grow exponentially, and scaling applications to use these extra cores is notoriously difficult. In such settings with spare cores, DoublePlay incurs only modest overhead.

For CPU-bound benchmarks, DoublePlay's offline replay takes approximately the same amount of time as a single-threaded execution of the application. This is due to our current implementation, which limits offline replay to executing on one core. If we used two executions to accelerate replay in the same way we accelerate recording, the offline replay time should be approximately the same as the record time for these benchmarks. Aget runs much faster during offline replay than during recording because it obtains its data from sequential disk reads rather than from network receives. Apache runs even faster because it uses Linux's zero-copy `sendfile`: thus, no data is copied into or out of its address space on replay. Mysql also benefits from sequential disk I/O.

DoublePlay's offline replay performance is a substantial improvement over system such as PRES and ODR that log partial information during recording, then search during replay for an execution that matches the original execution. In its low-recording overhead mode (SI-DRI), ODR's replay time is reported as ranging from 300 to over 39000 times the original application time, with some replays not completing at all. During the first replay of an execution, PRES records the global order of accesses to shared variables from different threads (called the RW scheme) to guide its search for an execution that matches the recorded run. This is reported to have an overhead from 28% (for network applications to several hundred times (for CPU-bound applications and benchmarks) the original execution time. This additional work is necessary to guarantee that the produced replay can be reproduced at will. Further, PRES may try several executions before finding a matching one. When PRES records synchronization operations and system calls (similar to DoublePlay), it takes 1-28 tries to replay most runs, and is unable to replay one run within 1000 tries. Once a valid replay is found, subsequent PRES replays do not incur this searching overhead. Thus, DoublePlay's contribution compared to these prior systems is (1) to guarantee that a replay can be produced in a bounded amount of time, and (2) to substantially lower relative replay time. The main cost of these contributions is that DoublePlay uses twice as many cores during recording to run two executions of CPU-bound applications.

6.3 Forward recovery and loose replay

Of the nine benchmarks we used in our evaluation, four (pbzip, aget, Apache, and mysql) experience application-level benign data races. For instance, pbzip2 has a benign data race in which an output thread spins waiting for a worker thread to set a value. Aget has a benign race where a thread reads and displays a progress counter without grabbing a lock. Apache has a benign race in which worker threads increment an idle counter in a spin loop with

an atomic compare-and-swap implemented using low-level locks. If two worker threads contend, one may experience an additional iteration of the spin loop, which leads to an additional paired lock-unlock sequence.

As shown in Table 2, DoublePlay's loose replay optimization reduces the frequency of rollbacks for pbzip2 — in fact, all rollbacks were eliminated during our evaluation. Eliminating rollbacks improves execution time from 6% of the original execution time with two threads to 13% with four threads. In all runs during our evaluation, the loose replay optimization was able to continue the execution of the epoch-parallel execution past the spin loop. While extra system calls create a transient difference in stack values, the process states soon converge when the thread overwrites those values during subsequent function calls.

For aget and Apache, no rollbacks were avoided. Although loose replay allows epoch-parallel execution to proceed when aget outputs a different progress value to the console, the output values usually remain in a buffer in `libc` at the end of the epoch, which is detected as a divergence. The data races in Apache lead to complicated divergences in synchronization operations and system calls for which loose replay cannot be used. Further, for these two applications, forward recovery does not have a measurable performance impact. For aget, the reason is that DoublePlay saves a copy of data received over the network until the epoch that received the data is committed (otherwise, the received data would be lost). On a rollback, the subsequent execution reads the copied data from memory rather than the network, so those system calls are much faster. Apache also benefits from this behavior, but it also takes frequent epochs because it sends external output over the network quite often. Consequently, the amount of work preserved by a forward recovery is very small. Since divergence check failures are relatively infrequent, the effect of forward recovery on the execution time of the two applications is negligible. We observed only one rollback during the mysql benchmarks; as with Apache, this rollback could not be avoided due to a complicated divergence, but the performance impact was negligible due to frequent epochs. Thus, while forward recovery and loose replay are often successful in reducing rollbacks and preserving work done during a failed epoch, they appear to be most beneficial for CPU-bound applications with longer epoch durations.

7. Conclusion

Providing efficient deterministic replay for multithreaded programs running on multiprocessors is challenging. While many prior solutions have been proposed, all fall short in some way. For instance, they may require custom hardware support, be prohibitively slow for many applications, or not guarantee that a replayed execution can be produced in a reasonable amount of time. Compared to these prior systems, DoublePlay's contribution is to provide guaranteed software-only record and replay with a minimal overhead to execution time, at the cost of using more cores. The key insight in DoublePlay is that one can use the simpler and faster mechanisms of single-processor record and replay, yet still achieve the scalability offered by multiple cores, by using an additional execution to parallelize the record and replay of an application. On machines with spare cores, this insight allows DoublePlay to record application execution with only an average of 15% overhead with 2 threads and 28% with 4 threads.

Acknowledgments

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

References

[1] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[2] BACON, D. F., AND GOLDSTEIN, S. C. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (1991), ACM Press, pp. 194–206.

[3] BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: safe multithreaded programming for c/c++. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (Orlando, Florida, USA, 2009), pp. 81–96.

[4] BHANSALI, S., CHEN, W., DE JONG, S., EDWARDS, A., AND DRINIC, M. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments* (June 2006), pp. 154–163.

[5] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[6] CHOI, J. D., ALPERN, B., NGO, T., AND SRIDHARAN, M. A perturbation free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium* (April 2001).

[7] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Technical Conference* (June 2008), pp. 1–14.

[8] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2009), pp. 85–96.

[9] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[10] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M., AND CHEN, P. M. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2008), pp. 121–130.

[11] FELDMAN, S. I., AND BROWN, C. B. IGOR: A system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (1988), pp. 112–123.

[12] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 2008 International Symposium on Computer Architecture* (June 2008), pp. 265–276.

[13] JR., R. L. B., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *OOPSLA* (2009), pp. 97–116.

[14] KELSEY, K., BAI, T., DING, C., AND ZHANG, C. Fast Track: A software system for speculative program optimization. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization (CGO)* (March 2009), pp. 157–168.

[15] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference* (April 2005), pp. 1–15.

[16] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Transaction on Computers 36*, 4 (1987), 471–482.

[17] LEE, D., SAID, M., NARAYANASAMY, S., YANG, Z. J., AND PEREIRA, C. Offline symbolic analysis for multi-processor execution replay. In *International Symposium on Microarchitecture (MICRO)* (2009).

[18] LEE, D., WESTER, B., VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 77–89.

[19] MELLOR-CRUMMEY, J. M., AND LEBLANC, T. J. A Software Instruction Counter. In *Proceedings of the 1989 International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 78–86.

[20] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 2008 International Symposium on Computer Architecture* (June 2008), pp. 289–300.

[21] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Recording shared memory dependencies using Strata. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[22] NARAYANASAMY, S., PEREIRA, C., PATIL, H., COHN, R., AND CALDER, B. Automatic logging of operating system effects to guide application-level architecture simulation. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)* (June 2006), pp. 216–227.

[23] NARAYANASAMY, S., POKAM, G., AND CALDER, B. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)* (June 2005), pp. 284–295.

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

[25] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[26] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, March 2008), pp. 308–318.

[27] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006), pp. 1–14.

[28] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2009), pp. 97–108.

[29] OPLINGER, J., AND LAM, M. S. Enhancing software reliability using speculative threads. In *Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 2002), pp. 184–196.

[30] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[31] PETERSON, Z. N. J., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transacations on Storage 1*, 2 (2005), 190–212.

[32] RONSSE, M., AND BOSSCHERE, K. D. RecPlay: A full integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (May 1999), 133–152.

[33] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), pp. 258–266.

[34] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[35] SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. Multiscalar processors. In *Proceedings of the 1995 International Symposium on Computer Architecture* (June 1995), pp. 414–425.

[36] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[37] STEFFAN, J. G., AND MOWRY, T. C. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 1998 Symposium on High Performance Computer Architecture* (February 1998), pp. 2–13.

[38] SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000), pp. 257–268.

[39] SÜSSKRAUT, M., KNAUTH, T., WEIGERT, S., SCHIFFEL, U., MEINHOLD, M., FETZER, C., BAI, T., DING, C., AND ZHANG, C. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization (CGO)* (April 2010), pp. 131–140.

[40] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. Triage: Diagnosing production run failures at the user's site. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (October 2007), pp. 131–144.

[41] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. quFiles: The right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (San Jose, CA, February 2010), pp. 1–14.

[42] VLACHOS, E., GOODSTEIN, M. L., KOZUCH, M. A., CHEN, S., FALSAFI, B., GIBBONS, P. B., AND MOWRY, T. C. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 271–284.

[43] WEERATUNGE, D., ZHANG, X., AND JAGANNATHAN, S. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the 2010 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2010), pp. 155–166.

[44] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June 1995), pp. 24–36.

[45] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 2003 International Symposium on Computer Architecture* (June 2003), pp. 122–135.

[46] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)* (June 2007).

[47] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 2010 European Conference on Computer Systems (EuroSys)* (April 2010), pp. 321–334.

[48] ZILLES, C., AND SOHI, G. Master/slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)* (2002), pp. 85–96.