Processor-Oblivious Record and Replay

ROBERT UTTERBACK*, Monmouth College, USA
KUNAL AGRAWAL and I-TING ANGELINA LEE, Washington University in St. Louis, USA
MILIND KULKARNI, Purdue University, USA

Record-and-replay systems are useful tools for debugging non-deterministic parallel programs by first ***recording*** an execution and then ***replaying*** that execution to produce the same access pattern. Existing record-and-replay systems generally target thread-based execution models, and record the behaviors and interleavings of individual threads. Dynamic multithreaded languages and libraries, such as the Cilk family, OpenMP, TBB, etc., do not have a notion of threads. Instead, these languages provide a ***processor-oblivious*** model of programming, where programs expose task parallelism using high-level constructs such as spawn/sync without regard to the number of threads/cores available to run the program. Thread-based record-and-replay would violate the processor-oblivious nature of these programs, as they incorporate the number of threads into the recorded information, constraining the replayed execution to the same number of threads.

In this paper, we present a processor-oblivious ***record-and-replay*** scheme for dynamic multithreaded languages where record and replay can use different number of processors and both are scheduled using work stealing. We provide theoretical guarantees for our record and replay scheme — namely that record is optimal for programs with one lock and replay is near-optimal for all cases. In addition, we implemented this scheme in the Cilk Plus runtime system and our evaluation indicates that processor-obliviousness does not cause substantial overheads.

ACM Reference Format:
Robert Utterback, Kunal Agrawal, I-Ting Angelina Lee, and Milind Kulkarni. 2019. Processor-Oblivious Record and Replay. *ACM Trans. Parallel Comput.* 1, 1 (October 2019), 27 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Debugging multithreaded programs is challenging, due to non-deterministic effects such as the interleaving of threads' accesses to shared data. Different thread interleavings can produce different results, and a bug that manifests under one interleaving may not manifest under another, making reproducing bugs notoriously difficult. A popular technique for addressing this problem is ***record and replay*** [3, 21, 30, 35, 37, 38, 42, 43, 45, 49, 56, 59, 61, 67, 75, 76, 78]. One execution ***records*** enough information about its behavior so that a second execution can faithfully ***replay*** that behavior, producing the same outcome. As a result, any bug that manifests during the recorded run will be reproduced during the replay run, easing the task of tracking down bugs.

In this work, we focus on programs where shared objects are protected by locks. A record-and-replay system for these programs must ensure that critical sections protected by the same lock are executed in the same order during the recorded run and the replay run. For standard

*This work was done in part while the author was affiliated with Washington University in St. Louis

Authors' addresses: Robert Utterback, rutterback@monmouthcollege.edu, Monmouth College, USA; Kunal Agrawal, kunal@wustl.edu; I-Ting Angelina Lee, angelee@wustl.edu, Washington University in St. Louis, USA; Milind Kulkarni, milind@purdue.edu, Purdue University, USA.

1539-9087/2019/10-ART $15.00
https://doi.org/0000001.0000001

thread-based programming models, this amounts to recording the order in which threads acquire each lock[1]. However, a class of parallel programming languages uses ***dynamic multithreading***, where the number of threads is not part of the model at all, such as the Cilk family [18, 34, 41], subsets of OpenMP [6], Threading Building Blocks [40], the Habanero family [7, 24], Task Parallel Library [47], X10 [24, 25], and many others. In these languages and libraries, the program itself is ***processor (or thread) oblivious*** — the programmer specifies the logical parallelism of the program using primitives such as spawn/sync, async/finish, or parallel-for loops. At run time, a scheduler is responsible for efficiently mapping this parallelism to ***worker*** threads that execute the computation in parallel.[2]

Despite the lack of explicit threads, record and replay is still useful for these dynamically multithreaded programs: if multiple parallel tasks access shared data using a lock, different executions might result in tasks accessing that data in different orders. These sources of non-determinism can lead to difficult-to-identify bugs. To our knowledge, there are no systems that perform record and replay for dynamic multithreading systems.

A straightforward approach to performing record and replay of a dynamic multithreaded program is to treat the composition of the program and the scheduler as a "standard" threaded program: the workers managed by the scheduler are the threads whose interleavings must be recorded. A clear drawback of this approach is that the *scheduler itself* is a highly parallel component: the workers are threads that access shared data (e.g., the queues of tasks that workers collaborate to complete). Cilk's scheduler, for example, uses ***randomized work stealing***: *which* workers execute *which* tasks *when* is non-deterministic and can change from one execution of a computation to the next even if the number of workers remains the same. Recording these computations using a standard record and replay scheme requires recording all of the decisions of the scheduler and then reproducing these exact decisions during replay, which can significantly increase the cost of both record and replay.

This paper is premised on this insight that *recording should occur at the correct level of abstraction.* Dynamic multithreading systems are deliberately processor oblivious: the programming and execution model make no reference to the number of threads. A record-and-replay system for dynamic multithreading should also be processor oblivious: by operating at the level of the programming model, rather than incorporating the scheduler into the recording domain, such a record-and-replay system can avoid the need to faithfully record and replay the behavior of the scheduler. Moreover, such a system would inherit the processor-oblivious nature of the underlying programming model: record and replay could use different numbers of workers, while delivering the same observed behavior.

In this paper, we present PORRidge, the first *processor-oblivious* record-and-replay system, and the first known record-and-replay system for dynamically multithreaded programs. PORRidge targets ***data-race free*** (DRF) Cilk programs — those whose accesses to shared data are correctly synchronized — and hence focuses on controlling the order in which synchronization operations are performed. While data race freedom may seem to be a strong constraint, we note two things. First, DRF is a common assumption for record-and-replay systems [35, 52, 67], as well as other dynamic analyses [58]. Second, DRF is a limitation of the PORRidge *implementation*, which needs to track sources of non-determinism. PORRidge uses the DRF assumption to allow it to track lock operations only. However, the same *conceptual* record and replay techniques could be applied to racy programs, by using race detection tools (e.g., [26, 31, 32, 46]) to identify races and indicate to

[1]If there are *race conditions* in the code, where shared data is not protected by a lock, additional steps must be taken to ensure safe recording of the accesses to that data. In this paper, we consider race-free programs.
[2]We use workers and processors interchangeably in this paper.

PORRidge additional sources of non-determinism (tools like Chimera adopt similar approaches [44]). We also assume that there is no parallelism within critical sections, which is a standard assumption for most dynamic multithreaded systems.

Following the processor-oblivious model, PORRidge is oblivious to the number of workers. Work stealing is used to schedule the computation during both record and replay. Hence, a program recorded on n workers can be replayed on m workers. Indeed, m can be greater than n — a program can be replayed on more processors than the original recorded run! Replaying on more processors than the recording can be useful during debugging: (i) debugging during replay can be performed with heavyweight instrumentation to aid in bug diagnosis, and replay on more processors can compensate for the additional overhead of instrumentation; (ii) if a bug is seen during recording long after a program has started, replay on more processors can reproduce the bug more quickly.

The key insight behind PORRidge is as follows: there are multiple sources of non-determinism in scheduling when we execute a dynamic multithreaded program, for instance, the random work stealing decisions that the scheduler makes. However, for a data-race free computation, a recording run need not record all this information to reproduce it faithfully during replay; it is sufficient to just record the order in which various critical sections acquired a shared lock. To be more precise, a dynamic multithreaded program can be viewed as a ***directed acyclic graph (DAG)***, with each node in the graph representing a task and edges between nodes represent dependencies. This graph is independent of the number of workers and for race-free computations, the only non-determinism arises from the order that tasks acquire locks. These lock acquisitions represent additional ***happened-before*** edges in the program DAG and recording these additional edges is sufficient to ensure that the DAG can be replayed faithfully.

Therefore, during a recording run, PORRidge simply records these happened-before edges. PORRidge thus sits on top of the Cilk runtime, and needs not track the fine-grained, non-deterministic interleaving of worker threads in the Cilk scheduler. Moreover, during the replay run, PORRidge ensures that the happened-before relationships that were recorded are respected: in other words, during replay, PORRidge schedules the ***augmented DAG*** which contains all these happened-before edges in addition to the original dependencies. While this new augmented DAG may have parallelism limited by the happened-before edges, its parallelism is *not* directly limited by the number of threads that the recorded run executed on.

Replay is more complex. The Cilk runtime system is designed to obey edges that arise from parallel primitives such as spawn/sync. Therefore, PORRidge adds mechanisms to the Cilk runtime system to respect the additional happened-before edges that arise from lock operations. However, these mechanisms, and generally all of the non-determinism of the scheduler, remain encapsulated separately from the replay itself. By keeping the runtime (both during record and during replay) outside the scope of the system, PORRidge is able to maintain low overhead.

Contributions

This paper makes several contributions:

(1) We present PORRidge, the first processor-oblivious record-and-replay system for dynamic multithreaded programs that keeps track of happened-before relationships between critical sections (Section 3). To our knowledge, this is the first record-and-replay system (processor oblivious or not) for these kinds of programs.

(2) We state and prove the theoretical guarantees for PORRidge (Section 4). Despite the fact that PORRidge requires additional happened-before tracking during record, and requires conforming to those happened-before edges during replay, it can provide strong guarantees. In particular, let W be the ***work*** required by a parallel computation — its serial execution time; let S be the ***span*** (or ***critical path length*** — longest sequence of dependencies in the

computation; let P be the number of processors; and let B be the amount of work in critical sections. Then, the runtime of the recorded execution is $O(W/P + S + B)$. For a single lock, this bound is *asymptotically optimal*. While replay incurs slightly higher costs due to the necessity of respecting happened-before edges, its runtime is $O(W/P' + S' \log P')$, where S' is the span of the augmented DAG (i.e., with the additional happened-before edges) and P' is the number of processors the *replayed* execution is run on. That means, it is possible for the replay to be *asymptotically faster* than the recorded run by using more processors.

(3) We implement a prototype of PORRidge by extending the Cilk Plus [41] runtime system and empirically evaluate our prototype system (Section 5). We show across six benchmarks that PORRidge delivers low overhead and good scalability for both record and replay. In particular, replay can often provide better speedup than record as we increase the number of cores. In addition, despite requiring additional runtime mechanisms in order to respect happened-before edges, the additional overhead of replay over the record is small.

(4) To evaluate the benefit of processor-oblivious record and replay, we implement a *second* record-and-replay system called PARRot that is processor-aware, and we empirically evaluate and compare its overhead to that of PORRidge (Section 6). Instead of using a generic record-and-replay system designed for multithreaded C code that will work out of the box, we tailor the design of PARRot to target a work-stealing runtime system to avoid overestimating the overhead of PARRot's approach. PARRot instruments and records only memory and lock accesses that can lead to nondeterministic scheduling choices, thereby incurring much less overhead than a generic record-and-replay system that records all shared memory and lock accesses. Despite our best efforts in performance engineering PARRot, it consistently incurs higher overhead than PORRidge and even fails to record one of the applications tested due to out-of-memory errors. In contrast, PORRidge was able to record and replay the same benchmark successfully with low overhead and obtain close to linear speedups, because it need not track all the nondeterminism in the runtime.

2 PRELIMINARIES

This section provides the necessary background on modeling parallel computations, work-stealing schedulers, and some definitions.

Dynamic Multithreading and Computation DAGs. We use Cilk Plus programming keywords, `cilk_spawn` and `cilk_sync`, to explain the dynamic multithreaded programming model; even though other languages use different keywords for creating and synchronizing tasks, they provide similar semantics. Parallelism is created using `cilk_spawn`. When a function instance F ***spawns*** another function instance G by preceding the invocation with `cilk_spawn`, the ***continuation*** of F — the statements after the spawning of G — may execute in parallel with G without waiting for G to return. A `cilk_sync` operation acts as a local barrier; the control flow cannot move past a `cilk_sync` in function F until functions previously spawned by F have returned.

As is common in the literature, we model the parallel computation as a ***directed acyclic graph (DAG)***, where each node is a unit-time computation and each edge represents a dependence between nodes. A particular node is ***enabled*** and becomes ***ready*** to execute when all its predecessors have executed. Also, as is common, we assume that each node has an out-degree of at most two. A ***strand*** is a chain of nodes all with in-degree and out-degree 1 — programmatically, a strand is a sequence of instructions that contain no parallel primitives and therefore must execute sequentially. We define two parameters on the DAG. ***Work*** W is the total number of nodes in the DAG and represents the execution time of the DAG on one processor. ***Span*** S is the length of the longest path and represents the execution time on an infinite number of processors.

Work-Stealing Scheduler. During execution, a ***work-stealing*** scheduler [19, 34] dynamically load balances a parallel computation across available ***worker*** threads. Each worker maintains a double-ended queue, or ***deque***, of available work (i.e., ready nodes). The node that is currently executed is called the ***assigned*** node. When a worker w finishes executing a node x, the following conditions may occur. If the execution of x enables one node, the enabled node becomes w's new assigned node. If it enables two nodes, one of them is placed on w's active deque and the other becomes the new assigned node. If it enables no node, w pops the ready node at the bottom of its deque and makes it its assigned node. If its deque becomes empty, w turns into a ***thief*** and chooses a ***victim*** worker uniformly at random to ***steal*** from. Given a computation with work W and span S, a work-stealing scheduler executes the computation in expected time $\frac{W}{P} + O(S)$ on P processors [19].

Modeling Critical Sections. Since our computations contain critical sections, we must model those. We assume that there is no parallelism within a critical section, and thus each critical section of length x is simply a strand (chain) of x unit time nodes in the DAG. Each node in the chain has in-degree one and out-degree one. The first node of this chain is called an *acquisition* node and the last node is called a *release* node. We say B_i is the total amount of time the lock ℓ_i is held — therefore, it is the sum of the lengths of all chains representing critical sections held by ℓ_i. We say that the ***total blocking time*** is $B = \sum_i B_i$.[3]

Augmented DAG. Once we record the execution of a computation DAG, we must replay it so that all the critical sections protected by the same lock are executed in the same order as the recorded execution. Therefore, additional happened-before edges are added to the computation DAG. We call the new DAG with the happened-before edges an ***augmented DAG***. More precisely, if a critical section b accesses lock ℓ_i after another critical section a that also accesses ℓ_i, with no other critical sections in between accessing ℓ_i, then we say that a is the ***predecessor*** critical section to b, and b is the ***successor*** critical section of a. In the augmented DAG, we add an edge from the last node (release node) of a to the first node (acquisition node) of b. (Note that since the last node of a has out-degree one from assumption, this still maintains the invariant that no node has out-degree greater than two). The work of this new DAG is still W since we haven't added new nodes. However, the span may be larger, and we denote the span of the augmented DAG by S'.

3 DESIGN OF PORRIDGE

We now describe the design of PORRidge. As mentioned in Section 1, since PORRidge is designed for data-race free programs, it needs to capture only the happened-before edges formed between critical sections protected by the same lock during recording and enforce the same happened-before edges during replay. Consequently, PORRidge has a light-weight recording process that can be implemented entirely outside of the work-stealing scheduler. The replay process, on the other hand, requires modifications to the work-stealing scheduler in order to guarantee the stated theoretical bound.

PORRidge provides wrappers for the various thread lock objects and associated acquisition/release functions; the wrapper functions are invoked via compile-time interpositioning [22][Chp. 7.13]. Each lock object in PORRidge contains a pointer to the underlying lock defined by the POSIX thread (Pthread) specification [39] and additional data structures to record and enforce happened-before edges.

[3]If we are working with racy programs where we instrument the racy accesses as light-weight critical sections, then we must also add those critical sections in the calculation of B.

3.1 Identifying Critical Sections

The information stored in each PORRidge lock must uniquely identify critical section instances in the computation DAG, and the identification must be consistent across different executions. Here, we use ***pedigree*** [48], a sequence of integers corresponding to the rank ordering of spawn statements in the ancestor functions (including the current function) that lead to the current strand. Pedigrees uniquely identify each strand in a consistent manner since they depend only on the computation DAG and not on the schedule. Critical section instances can be uniquely identified by uniquely identifying the strand they are in and then their ordering within the strand. Therefore, it is sufficient to modify the pedigree mechanism slightly to uniquely identify critical sections. Specifically, we append another integer to the pedigree sequence and increment this integer when a critical section ends, i.e., when a lock is released.

The open-source Cilk Plus runtime [41] readily provides support for pedigree; however, each read to the pedigree incurs a worst-case $\Theta(d)$ overhead, where d is the maximum ***spawn depth***, the number of spawn statements nested on the stack during serial execution. Since the pedigree must be read in every lock acquisition, this causes lock acquisitions to incur $\Theta(d)$ overhead during record and replay. Ideally, we would like the cost of lock acquisitions to be constant in order to guarantee both the record and replay time bounds.

To achieve the desired constant overhead, we use a strategy similar to DotMix [48] to give each critical section instance a ***critical section ID***, which is effectively a hash of the modified pedigree that can be maintained and derived with constant overhead. DotMix works as follows: the runtime generates a size-d vector of random numbers using a seed at the beginning of the computation. Given a pedigree, DotMix takes dot-product of the pedigree with the vector and mods the dot-product result with a large prime p; provided that we use the same seed, a pedigree always hashes to the same random number. Moreover, two random numbers generated via two different pedigree have a low probability of collision [48]. Using a similar strategy as DotMix, we obtain unique critical section IDs with constant overhead per lock acquisition by maintaining them incrementally — the runtime maintains and passes down the prefix of the dot-product calculation as the execution spawns each function. To obtain a critical section ID, a worker multiplies the rank ordering of the current strand with the appropriate element in the random vector, adds the product into the prefix, and mods the result. The prefix can be maintained with constant overhead per spawn, and the generation of the critical section ID given the prefix incurs constant overhead. The drawback of critical section IDs is the (rare) possibility of collisions, which we discuss in Section 3.2.

3.2 Record

Conceptually during recording, PORRidge stores with each lock an ordered list of previous acquisitions of the lock, henceforth referred to as the ***lock-acquisition list***. Upon lock creation, the lock is assigned an ID based on the current strand ID so that its lock-acquisition list can be assigned to it during replay. When a worker successfully acquires a lock, it simply adds the current critical section ID to the end of its lock-acquisition list. If the lock is not available, the worker spins[4].

As mentioned in Section 3.1, there is a (very) small possibility of collision where different critical section instances for the same lock yield the same critical section ID.[5] To handle these collisions, at each lock acquisition the critical section ID is also inserted into a per-lock hash table. The critical section ID is used as the key so that the hash table disambiguates lock acquisitions with the same

[4]PORRidge does not support `try-lock` operations but could be extended to handle them.

[5]Even though ID collisions can also occur among different critical section instances for different locks, such collisions do not matter since they are recorded in different lock-acquisition lists.

critical section ID. The value of each entry contains a pointer to the corresponding lock-acquisition list node. Upon a collision, the initial list node is marked as "has collision" and the pedigree of the current critical section is read and stored in the new lock-acquisition list node.

At the end of the recorded execution, every lock object writes out the critical section IDs (and pedigrees read in the event of collisions) in its lock-acquisition list to a log file in the order inserted.

3.3 Replay

At the beginning of the replay, the runtime reads in the previously recorded log and recreates the lock-acquisition list and hash table for each lock. When a lock is created the current strand ID is used to find the corresponding lock-acquisition list and hash table. PORRidge maintains the invariant that the head of the list always points to the next strand that should successfully acquire the given lock. Each list node also contains a pointer to the runtime data necessary to enable suspending and resuming the strand. During replay, if a worker encounters a lock acquisition for critical section b, and its predecessor — a lock release of the critical section a that was executed immediately before b during the recorded run — has not executed yet, the worker suspends the execution of the strand, since it is not ready in the augmented DAG. On the other hand, when some worker (in this case, the worker that executed a) releases a lock, it may enable critical section b (which was tried earlier and suspended); this worker must then resume this suspended critical section.

Lock Acquire. When a worker encounters a lock-acquisition operation, it must first find the lock-acquisition list node corresponding to the current critical section ID. This is done by searching the hash table with the current critical section ID. If duplicates are found, the full pedigree must be read and compared. If no pedigrees match, the worker concludes that its node is the first one, since no pedigree is read for the first critical section ID involved in a collision.

Once found, this list node is compared to the head of the corresponding lock-acquisition list. If the node is at the head, the worker acquires the lock and continues execution. Otherwise, the worker marks the corresponding list node to indicate that the lock-acquisition has been tried and suspended. It then suspends the execution of its current deque and work steals.

During process-oblivious replay, a worker must never spin or wait to acquire a lock — spinning not only hurts performance, it can also lead to deadlocks. Consider an execution recorded on multiple workers and replayed on one worker. Say the computation contains two critical sections x and y, protected by the same lock, that are logically in parallel with each other except for the happened-before dependence. If x comes before y during sequential execution, during replay on one worker, the worker will encounter x first. However, since record is done on multiple processors, y could have acquired the lock before x during record, and thus replay must execute y before it can execute x. If during replay, the worker simply spins when it encounters x, it will spin indefinitely since no other workers are around to execute y. Similar examples can also be created for multiple-processor replay. Therefore, when a worker encounters a lock that it cannot yet acquire during replay, it is essential that it suspends and finds other work to do.

Lock Release. The worker first advances the head of the list and checks to see if the next lock acquisition has been tried and suspended. If not, the worker simply continues the execution after the lock release. If the next lock acquisition has been tried and suspended, the worker performing the lock release now has two continuations that it can potentially work on — the continuation after the lock release, and the suspended lock acquisition enabled by this lock release. Both choices lead to the same theoretical guarantees. In our implementation, we chose to have the worker suspend the continuation after the lock release and resume the next lock acquisition in the list to reduce contention. Note that it is possible for a worker to release a lock while a different worker is concurrently suspending the next lock acquisition in line. Since there are at most two workers

concurrently operating on a given list node, we coordinate the synchronization using a variant of Dekker's algorithm [29], which coordinates the synchronization among two workers using atomic reads and writes.

Runtime Modifications. The fact that a lock acquisition causes a worker to suspend its current execution causes the PORRidge scheduler to diverge from the vanilla work-stealing scheduler used by Cilk Plus without locks. The vanilla work-stealing scheduler maintains the invariant that there are at most P deques containing ready work throughout execution, where P is the number of workers used, and this fact is important for proving the scheduling bound. The PORRidge scheduler no longer maintains the P-deque invariant, since a worker can suspend execution of a non-empty deque. Thus, the runtime must handle multiple deques per worker, and additional care must be taken to provide the provably-scalable time bound for replay.

During replay, a worker can suspend execution 1) upon a lock acquisition if the lock acquisition is not ready, or 2) upon a lock release, if the lock release in turn enables a suspended lock acquisition. In the first case, if the worker suspends its current (non-empty) deque, it work steals and allocates a new deque for the stolen work, thereby increasing the total number of deques. In the second case, the worker suspends the continuation after the lock release and resumes the deque containing the lock acquisition that it just enabled; in this case, the overall number of deques in the system does not increase.

One important thing to note is that a suspended lock acquisition is never on top of any deque and therefore no one ever steals it. When a worker suspends a deque due to a lock acquisition that is not ready, the suspended lock acquisition is at the bottom of the deque, and everything above it in the deque is ready. If the suspended deque contains only the lock acquisition, the PORRidge runtime frees the deque. The suspended lock acquisition, in turn, is always resumed (or enabled) by the lock release that unblocks it. In particular, if r and s are critical sections for the same lock, and r acquired the lock immediately before s during recording, then there is an edge from the lock release in r to the lock acquisition in s in the augmented DAG. Therefore, if the lock acquisition in s is suspended during replay, then s is resumed by the processor that executed the lock release in r. This ensures 1) that no worker ever waits or spins to acquire a lock, and 2) stealing into a suspended deque always results in a successful steal.

Since the PORRidge scheduler does not maintain the P-deque invariant during replay, we need to make a few changes to the scheduler to provide the provably good replay bound. First, we maintain the invariant that all P processors have approximately the same number of deques by the following mechanisms: (1) When a worker suspends a non-empty deque, it picks a worker uniformly at random to deposit the suspended deque with; and (2) on a steal attempt, if the thief steals into a victim worker v's a suspended deque and takes the last ready node in the deque, the deque is freed, and the thief randomly chooses a worker w and moves a suspended deque from w to v if w is not the same as v. Effectively, these mechanisms ensure that, whenever a deque appears in the system (due to suspension) or disappears from the system (due to stealing into the last ready node), the appearances and disappearances of deques are roughly equally distributed among workers to keep the loads of deques across workers roughly balanced. Second, given that all workers have approximately the same number of deques, we modify the steal protocol to ensure that workers steal from all deques uniformly at random. When a thief steals, it not only selects a victim at random, it also chooses among all the deques that the victim has to steal from at random. Finally, in the event that a worker releases a lock that enables both the continuation after the lock release and the next (suspended) lock acquisition, the worker suspends its deque with the continuation and executes the next lock acquisition by swapping its deque with the deque that contains the next lock acquisition. Doing so does not change the loads of deques across workers. We show how these changes allow us to provide a provably-scalable replay time bound in Section 4.

3.4 Performance Optimization

Thus far we have been discussing the design assuming each lock contains a lock-acquisition list and a hash table. The hash table can incur large overhead in practice for benchmarks that are already memory-bound (such as the graph benchmarks described in Section 5), since random accesses to the hash table inherently lack locality and incur additional cache misses. We optimized the implementation of the record phase in PORRidge by using a small Bloom filter [16] to detect critical section ID collisions in place of a hash table. Doing so allows the PORRidge to store the Bloom filter with the lock object itself, leading to better spatial locality, and it uses much less space than keeping an actual hash table. The trade-off is that a Bloom filter can report false positives (i.e., detecting collisions between section IDs with different values) and thus may lead to reading and logging the full pedigrees unnecessarily. In our experiments, however, we find that using the Bloom filter outperforms the hash table due to cache effects.

Even though we were able to use a Bloom filter during recording, the same optimization does not work during replay. During replay, beyond collision detections, a worker encountering a lock acquisition that is not yet at the head of the list needs to find the corresponding list node in order to mark it suspended. A Bloom filter is not sufficient for such a purpose. Since the number of lock acquisitions for a given lock is known a priori during replay, PORRidge uses a simple array to store the lock acquisitions instead. Moreover, whether a particular lock acquisition incurs a collision or not is also known. At a lock acquisition, either the lock acquisition is at the head, or if there is a possibility of collision and suspension, a worker scans the entirety of the array (starting at the current head) to check for collisions of its critical section ID and to find the corresponding list node. For many benchmarks, the number of acquisitions for a given lock is small; thus, the spatial locality and decrease in memory usage when using a simple array outweighs the benefit of constant-time search via a hash table.

Finally, in the PORRidge scheduler, we perform the following optimization that is not necessary for the scheduling bound but which improves performance in practice. Recall that there are two types of suspended deques. (1) When a worker suspends a deque due to a lock acquisition, the bottom node of this deque is a suspended node. (2) When a worker suspends a deque after a lock release, all the nodes in the deque are ready to resume and there are no suspended nodes. When another worker steals from the second type of deque with no suspended nodes, instead of stealing just the top strand, it ***mugs*** the entire deque and resumes the bottom node. This optimization reduces the number of deques faster, making replay more efficient.

4 THEORETICAL ANALYSIS

In this section, we prove theoretical upper bounds on the running time of our record and replay strategy. Recording and replaying are done in different processes and we provide separate bounds for them. The bound on the record process follows directly from the bounds for work-stealing. For replay, we analyze the scheduling strategy provided in Section 3. In the analysis, we extensively use the analysis of work stealing using a potential function provided by Arora, Blumofe and Plaxton [4] (abbreviated as ABP).

4.1 Running Time of Record

Theorem 4.1. *Given a computation with work W, span S, and blocking B (defined in Section 2), if we record the computation on P processors, the running time is $O(W/P + S + B)$ in expectation.*

Record analysis directly follows from work-stealing analysis. The only additional factor is that a worker spins when waiting on a lock, making no progress towards completing the computation.

Thus, we divide the computation into two types of phases and bound them separately. A phase is ***non-blocking*** if no processor is waiting on a lock, otherwise it is ***blocking***.

Lemma 4.2. *The total amount of time spent in blocking phases is at most B.*

Proof. This statement simply follows from the fact that the total time any processor could be holding the lock is at most B. □

Lemma 4.3. *The total expected time spent in non-blocking phases is $W/P + O(S)$. The time spent in non-blocking phases is $W/P + O(S + \lg 1/\epsilon)$ with probability $1 - 1/\epsilon$.*

Proof. During non-blocking phases, the processors are either working or stealing. The total number of work steps is at most W, since each work step consumes a unit of work in the computation DAG. From an argument very similar to that in ABP [4], one can show that the total number of steal steps when no worker is blocked is $O(PS)$ in expectation and $O(PS + P \lg 1/\epsilon)$ with probability at least $(1 - 1/\epsilon)$. Since there are a total of P processors executing these work or steal steps, the total time spent on non-blocking phases is as stated. Note that some work may also be done during blocking phases; however, this only over-estimates the running time. □

Combining Lemmas 4.2 and 4.3 gives us the stated theorem.

4.2 Running Time of Replay

Theorem 4.4. *Given an augmented DAG with work W and span S', the replay process completes in expected time $O(W/P + S' \lg P)$.*

As with the analysis of recording, we divide replay time steps into work steps and steal steps. No worker ever waits on a lock, so there are no blocking steps. The total work is still bounded by W. Therefore, it only remains to bound the number of steal attempts.

We use the ideas from the ABP analysis to bound the number of steal attempts. The main difference between vanilla work stealing and our replay strategy is that we now have more than P deques. In particular, the high-level idea in the ABP analysis is the following. If there are X deques in the system, then X steal attempts are likely to reduce the critical path by a constant amount. Therefore, the total number of steal attempts is ((number of deques) $\times S$) in expectation. Since our scheduler can have an arbitrarily large number of deques (as large as the number of critical sections in the program), we would get a very bad bound if we directly applied that technique. We use additional insights to bound the number of steal attempts for a replay scheduler. We first make the following observation.

Observation 1. *A steal from a suspended deque always succeeds since it is never empty. Since a successful steal is followed by a unit of work by the thief, the total number of steals from suspended deques is bounded by W.*

Note also that when the number of suspended deques is small, i.e. still approximately $O(P)$, we can use an analysis similar to ABP to bound the steal attempts. We only run into issues when the number of suspended deques is not small. Thus, in the analysis, we divide the phases into two different types.

A ***work-bounded phase*** begins when at least $P/2$ workers have at least one suspended deque. During a work-bounded phase, about a quarter of the steal attempts are likely to succeed (since that many steals occur from a suspended deque). Thus we can bound the total number of steal attempts in these phases by the work of the computation. A ***steal-bounded*** phase begins with fewer than $P/2$ workers having any suspended deques. Recall, as described in Section 3, we try to keep the number of deques across workers roughly balanced by throwing deques to workers

at random. Therefore, if fewer than $P/2$ workers have suspended deques, the total number of deques in the system are likely to be small. Therefore, we will bound the steal attempts occurred during steal-bounded phases using analysis similar to that in ABP. Note that each phase is either work-bounded or steal-bounded.

We can now see that if there are sufficient deques in the system, then the phase will be work bounded.

Lemma 4.5. *Say there are more than $2P$ suspended deques. At least $P/2$ workers have at least one suspended deque with probability at least $1 - (e/8)^{P/2} \geq 1 - 1/P^2$ for large enough P.*

Proof. The probability that $P/2$ workers have no suspended deques is $\binom{P}{P/2}(1/2)^{2P} \leq (2e/16)^{P/2}$. □

In addition, we can bound the number of steal attempts in a work bounded phase by the work of the computation.

Lemma 4.6. *The expected number of steal attempts during work-bounded phases is $O(W)$.*

Proof. In work bounded phases, at least $P/2$ processors have suspended deques. Since a thief chooses a victim uniformly at random, we have $1/2$ probability of stealing into these processors with suspended deques. In addition, since these workers have at most one active deque and at least one suspended deque, about half of the steals from these workers are expected to be successful. Therefore, the expected number of steals attempts during work-bounded phases is $4X$ where X is the number of steal attempts from suspended deques. Combining with Observation 1 gives the lemma. □

Therefore, we now only need consider the phases where the total number of deques in the system is smaller than $2P$. We now consider bounding the steal attempts in steal-bounded phases. Although we now potentially have more than P deques, we can still use analysis similar to ABP to bound the steal attempts. At a very high level, the ABP analysis works as follows. The computation starts out having bounded "potential," which is a function of the computation's span. Note that the important node that one needs to execute in order to make progress on the span always sits on top of some deque. The key point in the ABP analysis is that, if there are X deques in the system, and we steal uniformly at random from them, then after $O(X)$ steal attempts, some worker steals and executes the important node at the top of some deque and thus make progress on the span. Hence we can bound the number of steal attempts to be $O(XS)$ in expectation.

Similar to ABP, we define a potential function based on the depth of nodes in the augmented DAG. The depth of a node $d(u)$ is recursively defined as one plus the maximum depth of all its parents. The weight of a node is $w(u) = S' - d(u)$. Then, we define a potential as follows:

Definition 4.7. The ***potential*** $\Phi(u)$ of a node u is $3^{2w(u)-1}$ if u is assigned, and $3^{2w(u)}$ if u is ready.

The total potential of the computation is the sum of the potentials of all its ready and assigned nodes, and the follow lemma follows from the ABP analysis in a straightforward manner.

Lemma 4.8. *The initial potential is $3^{2S'-1}$ and it never increases during the computation.* □

The following lemma is a straightforward generalization of Lemmas 7 and 8 in ABP [4].

Lemma 4.9. *Let Φ_i denote the potential at time t and say that the probability of each deque being a victim of a steal attempt is* at least *$1/X$. Then after X steal attempts, the potential is at most $\Phi(t)/4$ with probability at least $1/4$.* □

In ABP, X would be P. In our case, we need to analyze what X is. To bound X, we define the number of suspended deques a worker has as its ***load***, and we are concerned with the ***maximum load***, i.e., highest number of suspended deques a worker can have.

Lemma 4.10. *If there are at most $K \leq P \lg P$ suspended deques over all processors, then with probability at least $1 - o(1/P)$, the processor with the largest load has at most $k = \lg P + O(1)$ suspended deques.*

Proof. The lemma follows from the classic balls into bins analysis [54, 62] — if we throw K balls into P bins, the number of balls in the most loaded bin is $O(1) + \lg P$ if $K < P \lg P$. If we think of suspended deques as balls and processors as bins, by performing the load balancing of suspended deques described at the end of Section 3, this result guarantees that when deques are suspended, they are distributed approximately evenly. □

In steal bounded phases, $K \leq 2P \leq P \lg P$. Therefore, the maximum load is at most $\lg P$ with high probability. We will say that a distribution is ***balanced*** if the processor with the largest number of deques has fewer than $2 \lg P$ deques, otherwise, we will say that the distribution is unbalanced.

We divide each steal bounded phase into rounds with $2P \lg P$ steal attempts. We say that a round is ***good*** if the maximum load is at most $2 \lg P$ throughout the round and bad otherwise.

Lemma 4.11. *Let $\Phi(t)$ denote the potential at the beginning of a good round. After $2P \lg P$ steal attempts, at the end of the round, the potential is at most $3\Phi(t)/4$ with probability at least $1/4$.* □

Proof. Since the maximum load is at most $2 \lg P$ throughout the round (by the definition of a good round), the probability that a particular steal attempt hits a particular deque is at least $1/(2P \lg P)$ (it may be higher since some workers have fewer than $\lg P$ suspended deques). Therefore, we can apply a small modification to Lemma 4.9 generalized from ABP and argue that the total potential decreases. Specifically in ABP, X from Lemma 4.9 would be P; in our case, X is $2P \lg P$. □

Lemma 4.12. *The total number of good rounds is $O(S')$ in expectation.*

Proof. At a high level, from Lemma 4.11, a constant number good rounds suffice to decrease the potential by a constant factor in expectation. Therefore, the number of rounds needed to reduce the potential to one is log of the initial potential, which is $3^{2S'}$. Therefore, after $O(S')$ rounds, the potential disappears and the computation completes. This argument of how we bound the number of good rounds based on reducing the potential is similar to how the ABP analysis bounds the number of phases. □

We still need to bound the number of bad rounds, however.

Lemma 4.13. *The number of bad rounds is $O(X)$ where X is the number of good rounds.*

Proof. A steal bounded round is good with probability at least $1 - o(1)$ from Lemma 4.10. Therefore, the total number of bad rounds is smaller than the total number of good rounds. □

The following lemma follows from Lemmas 4.12 and 4.13 and the fact that each round has $P \lg P$ steals.

Lemma 4.14. *The expected number of steal attempts in steal bounded phases is at most $O(S'P \lg P)$.*

The following lemma follows from Lemmas 4.6 and 4.14

Lemma 4.15. *The total number of steal attempts across all phases is $O(W + S'P \lg P)$.*

Lemma 4.15 and the facts that the total amount of work is W implies Theorem 4.4.

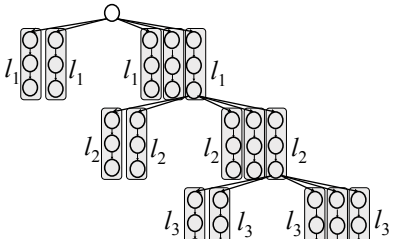

Fig. 1. An example DAG with multiple locks where getting a tight bound for recording is impossible for an online scheduler. The offline scheduler can always schedule the "important" (in this case, the right-most) critical section first, but an online scheduler has no way of knowing which critical section is "important", and therefore may execute it last.

4.3 Discussion

We now discuss how good or bad these bounds are, theoretically. For a single lock, note that W/P, S, and B are all lower bounds on the execution on P workers; therefore, the bound is tight. For multiple locks while W/P and S are still lower bounds, B is not a lower bound for all DAGs. Nevertheless, this bound is existentially tight — there exist DAGs for which it is tight. In general, it is difficult for online schedulers to get tight bounds on all computation DAGs with multiple locks without knowing what the future DAG looks like. Consider the DAG shown in Figure 1. Gray rectangles represent critical sections, and all critical sections in the same layer access the same lock. An optimal offline scheduler will schedule the right-most critical section of each layer first so it can schedule the next layer in parallel with the previous layers and can get good speedup. However, an online scheduler cannot know which critical section of each layer leads to more future work and may execute them in an order that gets no speedup. In general, an online scheduler cannot guarantee optimality, since for any online strategy, there is a bad DAG where the next layer is always created by the critical section this strategy executes last.

Let us now consider replay. In this case, W/P, and S' are lower bounds; therefore the replay bound of $O(W/P + \lg PS')$ is nearly tight — it just has an additional $\lg P$ factor on the span term which is tiny for most machines. In addition, since it is on the span term, according to the work-first principle [34], this overhead does not affect computations with sufficient parallelism.

On series-parallel (or more generally, fully-strict) computations, depth-first work stealing (of the kind we use) also guarantees a space bound; in particular, if the sequential execution uses S_1 stack space, work-stealing uses $O(PS_1)$ when using P workers. Since record uses vanilla work-stealing, it also provides this space bound. However, the replay scheduler executes the augmented DAG which is not a fully-strict DAG, and thus such a space bound is unattainable, as indicated by the lower bound in Section-3.1 in [17].

5 EMPIRICAL EVALUATION OF PORRIDGE

This section presents an empirical evaluation of PORRidge. The main benefit of a processor-oblivious record-and-replay system is that one can replay an execution on a different number of processors from that used during the recording — including a larger number — allowing the replay to benefit from parallel execution. There are inherent overheads in the record and replay in order to allow processor-oblivious replay, however. Specifically, during record, PORRidge must record happened-before edges via critical section IDs in a schedule-independent fashion; during replay, PORRidge may need to suspend and resume strands upon lock acquisitions and releases.

	number	*number of lock acquisitions*				
application	*of locks*	*total*	*min*	*max*	*mean*	*std. dev.*
`chess`	4	2.8e4	0	2.8e4	7.1e3	1.4e4
`dedup`	1	7.3e5	7.3e5	7.3e5	7.3e5	n/a
`ferret`	1	256	256	256	256	n/a
`matching`	5e6	5e7	5	25	10	2.23
`MIS`	5e6	2.8e6	3	27	5.63	2.73
`refine`	4.8e7	1.2e7	0	27	0.26	0.56

Table 1. Application benchmarks used and their execution characteristics measured when running on one worker. The total column shows the total number of lock acquisitions across all locks during execution. The min column shows the minimum number of lock acquisitions invoked on a given lock across all locks; similarly, the max column shows the maximum. The last two columns show the average number of lock acquisitions per lock and the standard deviation. The numbers of lock acquisitions when running on multiple workers are the same for `chess`, `dedup`, and `ferret` but can differ slightly (albeit close) for `matching`, `MIS`, and `refine` due to nondeterminism in the computations.

We empirically evaluated the overhead and scalability of the record phase and replay phase across six benchmarks with different execution characteristics. Our results indicate that, for benchmarks that have a sufficiently large work-to-critical-section ratio, record and replay incur negligible overhead. For benchmarks whose work is dominated by critical sections, record and replay can incur up to 3.73× overhead, with replay incurring slightly higher overhead than record. In terms of scalability, recording scales similarly compared to the baseline. As long as there is sufficient parallelism in the recorded execution, the replay scales similarly. Moreover, due to its non-blocking execution model, the replay continues to get speedup beyond P_{rec} workers, where P_{rec} is the number of workers used during recording.

Benchmarks. We used the following six benchmarks to evaluate the PORRidge system. The first one, `chess`, is a Cilk Plus program published by Intel [66] that solves a chess puzzle — given eight chess pieces excluding pawns, it counts the number of configurations where the pieces can attack all squares on an 8×8 chess board. The original program uses reducers [33] to keep the count of the number of such configurations found and to perform I/O; we modified the program to use locks instead. Two benchmarks, `dedup` and `ferret`, are from the PARSEC benchmark suite [13, 14]; they can be implemented as Cilk Plus programs that use locks for performing file I/O. Finally, we converted several nondeterministic versions of graph algorithms from the Problem Based Benchmark Suite [70] to use locks instead of Compare-And-Swap (CAS): `MIS` (Maximal Independent Set), `matching` (Maximal Matching), and `refine` (Delaunay Refinement). These benchmarks cover a wide spectrum of behaviors. Their runtime characteristics when executing on one worker are shown in Table 1. Note that the characteristics during parallel execution may differ slightly for some of the graph benchmarks as they are nondeterministic by nature. The first three benchmarks use few locks, but still have plenty of critical sections; however, they do a significant amount of work outside of critical sections. The graph benchmarks use a much larger number of locks, since there is one lock per vertex in the input graph. In addition, they do almost all of their work within critical sections.

Experimental Platform. We ran our experiments on an Intel Xeon E5-2665 with 16 2.40-GHz cores on two sockets; 64 GB of DRAM; two 20 MB L3 caches, each shared among 8 cores; and private L2- and L1-caches of sizes 2 MB and 512 KB, respectively. Both hyperthreading and dynamic frequency scaling are disabled in order to get consistent results across runs. For recorded runs, running times are in seconds as the mean of five runs, and we used a 64-bit Bloom filter in our implementation (see Section 3). For a given number of workers, the recording with the median

application	*baseline*	*record*	*replay*
`chess`	64.43	64.38 (1.00×)	65.11 (1.01×)
`dedup`	48.04	48.20 (1.00×)	48.16 (1.00×)
`ferret`	8.92	8.89 (1.00×)	9.10 (1.02×)
`matching`	3.06	9.64 (3.15×)	10.07 (3.29×)
`MIS`	1.01	3.42 (3.39×)	3.77 (3.73×)
`refine`	11.70	14.73 (1.26×)	13.63 (1.16×)

Table 2. Execution times running on one worker ($P_{base} = P_{rec} = P_{rep} = 1$) for six benchmarks, in seconds. The `replay` column shows the replay execution time for replaying the run recorded with one worker. The numbers shown in parenthesis indicate the overhead compared to the baseline.

running time is chosen for the replay runs. For the most part, the standard deviation was within 5% of the mean for both record and replay. A few data points were the exception — graph algorithms that are memory-bound (`matching` and `MIS`) have higher standard deviation during some replay, up to 12% for `MIS`.

Notation. We use the following notations in this section. The label ***baseline*** refers to executions of the benchmarks with ordinary spin locks (i.e., without PORRidge). The label ***record*** refers to the executions with recording enabled using PORRidge. The label ***replay*** refers to the executions with replay enabled using PORRidge. We use P_{base} to refer to the number of workers used during baseline execution, P_{rec} to refer to the number of workers used during record and P_{rep} to refer to the number of workers used during replay.

5.1 Overhead of Record

To evaluate the recording overhead, we compare the running time of PORRidge recording on one worker with the baseline running on one worker. Table 2 shows the execution times of six benchmark for these configurations. The recording overhead ranges from 1–3.39× with a geometric mean of 1.62×. Since PORRidge incurs overhead only upon lock operations, the overhead is in part dictated by how much work is done per lock acquisition. For programs that perform sufficient amount of work outside of critical sections, such as `chess`, `dedup`, and `ferret`, the overhead is negligible. The graph algorithms, especially `matching` and `MIS`, incur higher overhead. For these applications, almost all of the work occurs inside critical sections. In addition, each critical section does a very small amount of work. Their executions mostly involve repeatedly traversing some edge, acquiring a lock corresponding to the vertex at the end of the edge, updating a field in the vertex, and releasing the lock. Hence, the execution time of these programs is dominated by the cost of acquiring and releasing locks. Moreover, these applications are memory bound — they have large working sets and display very little locality in accessing data. The additional space used for logging during recording puts additional pressure on the memory hierarchy. In the initial implementation, we have used a hash table to detect collisions of critical section IDs (discussed in Section 3), and the additional cache misses incurred by accessing the hash table incurred much larger overhead in these applications (8–9×). By reordering the book-keeping data layout to obtain better spatial locality and using a Bloom filter instead of a hash table, we were able to reduce the overhead drastically.

5.2 Overhead of Replay

Replay has two types of overheads. Replay, like record, incurs overhead upon lock acquisitions and releases. When a worker tries to acquire a lock, it must search the lock-acquisition array with the current critical section ID to see if this lock acquisition is the next in line to obtain the lock. If so, it can acquire the lock. Otherwise, it must suspend. Upon release, the worker advances the head

	replay on one, recorded on P					
application	*P = 1*	*P = 2*	*P = 4*	*P = 8*	*P = 12*	*P = 16*
chess	65.14 (1.01×)	65.13 (1.01×)	65.11 (1.01×)	65.17 (1.01×)	65.20 (1.01×)	65.13 (1.01×)
dedup	48.16 (1.00×)	48.15 (1.00×)	48.16 (1.00×)	48.21 (1.00×)	48.11 (1.00×)	48.20 (1.00×)
ferret	9.10 (1.02×)	8.95 (1.01×)	8.95 (1.01×)	8.94 (1.01×)	8.93 (1.00×)	8.93 (1.00×)
matching	10.07 (1.04×)	10.13 (1.05×)	10.13 (1.05×)	10.17 (1.05×)	10.37 (1.08×)	10.43 (1.08×)
MIS	3.77 (1.11×)	3.90 (1.15×)	3.94 (1.16×)	3.93 (1.16×)	3.97 (1.17×)	4.04 (1.19×)
refine	13.63 (0.93×)	13.67 (0.93×)	13.47 (0.91×)	13.73 (0.93×)	13.70 (0.93×)	13.73 (0.93×)

Table 3. Execution times, in seconds, when replaying on one worker executions recorded on different number of workers. The numbers shown in parenthesis indicate the overhead compared to the execution time of that recorded on one worker.

of the lock-acquisition list; if the next lock acquisition has been tried and suspended, the worker suspends its current execution and resumes the execution of the next lock acquisition. In addition, replay also incurs overheads due to maintaining more deques than the vanilla Cilk runtime system.

If we record on one worker and replay on one worker, the record and replay executions proceeds in exactly the same order. Therefore, the replay execution never has to suspend. Essentially, the work done by replay is the same as the work done by record except that replay may need to search through the lock-acquisition arrays if there is a collision in the critical section IDs. Thus, for most benchmarks, replay exhibits similar overhead as in recorded run when $P_{rec} = P_{rep} = 1$ as shown in Table 3.

The more interesting case is when we record on more than one worker and replay on one worker ($P_{rec} > 1$ and $P_{rep} = 1$). In this case, process-oblivious replay has additional overheads — namely the overhead of suspending and resuming lock acquisitions. Note that when we replay (on one worker) an execution recorded on multiple workers, the worker likely encounters critical sections in a different order than the recorded execution did. When a worker encounters a critical section that it cannot execute yet, it must suspend its current deque and work steal. In addition, since it steals work at random, the next critical section it acquires may again not be ready. Therefore, the worker may suspend many deques before encountering a critical section it can execute.

We can gauge such an overhead by comparing the overhead of executions with $P_{rec} > 1$ and $P_{rep} = 1$ with the overhead of executions with $P_{rec} = P_{rep} = 1$ shown in Table 3. It turns out that for the most part, the additional overhead incurred by suspending and resuming lock acquisitions is negligible — at most a few percent increase for the memory-bound benchmarks.

5.3 Scalability of Record

To analyze the scalability of PORRidge for recording, we compare the speedup of record to the baseline's. The speedup is computed with respect to their respective one-worker execution counterpart. Table 4 shows scalability of both the baseline and recorded runs across benchmarks (the first two columns). The scalability profile for record tracks that of the baseline closely across all benchmarks. This is especially surprising for memory-bound benchmarks since the workers spend longer within critical sections during recording compared to the baseline. In spite if this, it appears that the additional overhead is distributed across processors evenly and did not reduce the overall parallelism by much.

5.4 Scalability of Replay

Table 4 also shows scalability of replay runs that replay executions recorded on $P_{rec} = 1, 2, 4, 8, 12, 16$ processors. Here, we measure the speedup of a replay run by comparing it against the time replaying the same recorded execution on one worker (i.e., $P_{rec} = 1$).

				replay on P workers ($P_{rep} = P$) an execution recorded on P' workers ($P_{rec} = P'$)					
application	*P*	*baseline*	*record*	*P' = 1*	*P' = 2*	*P' = 4*	*P' = 8*	*P' = 12*	*P' = 16*
chess	1	64.49 (1.00×)	64.36 (1.00×)	65.14 (1.00×)	65.13 (1.00×)	65.11 (1.00×)	65.17 (1.00×)	65.20 (1.00×)	65.13 (1.00×)
	2	32.20 (2.00×)	32.20 (2.00×)	32.60 (2.00×)	32.60 (2.00×)	32.61 (2.00×)	32.64 (2.00×)	32.63 (2.00×)	32.66 (1.99×)
	4	16.11 (4.00×)	16.11 (4.00×)	16.50 (3.95×)	16.36 (3.98×)	16.35 (3.98×)	16.37 (3.98×)	16.35 (3.99×)	16.34 (3.99×)
	8	8.15 (7.91×)	8.13 (7.92×)	8.55 (7.62×)	8.31 (7.84×)	8.42 (7.73×)	8.45 (7.71×)	8.34 (7.82×)	8.31 (7.84×)
	12	5.38(11.99×)	5.38(11.96×)	5.91(11.02×)	5.65(11.53×)	5.75(11.32×)	5.75(11.33×)	5.66(11.52×)	5.63(11.57×)
	16	4.04(15.96×)	4.11(15.66×)	4.52(14.41×)	4.50(14.47×)	4.47(14.57×)	4.57(14.26×)	4.32(15.09×)	4.35(14.97×)
dedup	1	48.04 (1.00×)	48.20 (1.00×)	48.16 (1.00×)	49.15 (1.00×)	48.16 (1.00×)	48.21 (1.00×)	48.11 (1.00×)	48.20 (1.00×)
	2	24.43 (1.87×)	24.44 (1.94×)	24.58 (1.92×)	24.44 (1.88×)	24.44 (1.95×)	24.43 (1.95×)	24.45 (1.96×)	24.42 (1.94×)
	4	12.43 (3.86×)	12.40 (3.89×)	12.61 (3.82×)	12.55 (3.84×)	12.51 (3.85×)	12.40 (3.89×)	12.40 (3.88×)	12.41 (3.88×)
	8	6.43 (7.47×)	6.49 (7.43×)	6.72 (7.17×)	6.69 (7.20×)	6.61 (7.29×)	6.46 (7.47×)	6.45 (7.46×)	6.46 (7.46×)
	12	4.52(10.63×)	4.53(10.64×)	4.88 (9.87×)	4.83 (9.97×)	4.75(10.14×)	4.63(10.41×)	4.55(10.57×)	4.55(10.59×)
	16	3.61(13.31×)	3.65(13.32×)	3.95(12.19×)	3.94(12.22×)	3.89(12.38×)	3.76(12.82×)	3.69(13.04×)	3.64(13.24×)
ferret	1	8.92 (1.00×)	8.89 (1.00×)	9.10 (1.00×)	8.95 (1.00×)	8.95 (1.00×)	8.94 (1.00×)	8.93 (1.00×)	8.93 (1.00×)
	2	4.52 (1.97×)	4.57 (1.95×)	4.53 (2.01×)	4.54 (1.97×)	4.56 (1.96×)	4.52 (1.98×)	4.53 (1.97×)	4.52 (1.98×)
	4	2.31 (3.86×)	2.33 (3.82×)	2.32 (3.92×)	2.34 (3.82×)	2.32 (3.86×)	2.32 (3.85×)	2.31 (3.87×)	2.32 (3.85×)
	8	1.27 (7.02×)	1.24 (7.17×)	1.24 (7.34×)	1.24 (7.22×)	1.27 (7.05×)	1.26 (7.10×)	1.26 (7.09×)	1.26 (7.09×)
	12	0.91 (9.80×)	0.91 (9.77×)	0.93 (9.78×)	0.90 (9.94×)	0.91 (9.84×)	0.92 (9.72×)	0.92 (9.71×)	0.91 (9.81×)
	16	0.76(11.74×)	0.78(11.40×)	0.75(12.13×)	0.75(11.93×)	0.75(11.93×)	0.75(11.92×)	0.75(11.91×)	0.74(12.07×)
matching	1	3.06 (1.00×)	9.64 (1.00×)	10.07 (1.00×)	10.13 (1.00×)	10.13 (1.00×)	10.17 (1.00×)	10.37 (1.00×)	10.43 (1.00×)
	2	1.92 (1.67×)	5.78 (1.46×)	6.88 (1.49×)	6.82 (1.49×)	6.93 (1.46×)	6.64 (1.53×)	6.68 (1.55×)	6.63 (1.57×)
	4	0.96 (3.19×)	3.03 (3.18×)	3.95 (2.55×)	3.90 (2.60×)	3.75 (2.70×)	3.64 (2.79×)	3.63 (2.86×)	3.62 (2.88×)
	8	0.50 (6.12×)	1.68 (5.74×)	3.77 (4.31×)	3.99 (4.57×)	4.22 (4.61×)	2.22 (4.58×)	2.21 (4.69×)	2.11 (4.94×)
	12	0.32 (9.56×)	1.13 (8.53×)	2.57 (3.92×)	2.42 (4.19×)	2.21 (4.58×)	1.89 (5.38×)	1.77 (5.86×)	1.68 (6.21×)
	16	0.25(12.24×)	0.89(10.83×)	2.56 (3.93×)	2.41 (4.20×)	2.11 (4.80×)	1.79 (5.68×)	1.58 (6.56×)	1.57 (6.64×)
MIS	1	1.02 (1.00×)	3.40 (1.00×)	3.77 (1.00×)	3.90 (1.00×)	3.94 (1.00×)	3.93 (1.00×)	3.97 (1.00×)	4.04 (1.00×)
	2	0.65 (1.57×)	2.03 (1.67×)	2.57 (1.47×)	2.54 (1.54×)	2.48 (1.59×)	2.47 (1.59×)	2.45 (1.62×)	2.52 (1.60×)
	4	0.32 (3.19×)	1.03 (3.30×)	1.63 (2.31×)	1.52 (2.57×)	1.36 (2.90×)	1.34 (2.93×)	1.33 (2.98×)	1.32 (3.06×)
	8	0.16 (6.38×)	0.58 (5.86×)	1.38 (2.73×)	1.14 (3.42×)	0.93 (4.24×)	0.79 (4.97×)	0.81 (4.90×)	0.79 (5.11×)
	12	0.13 (7.85×)	0.22 (8.10×)	1.54 (2.45×)	1.26 (3.10×)	0.89 (4.43×)	0.69 (5.70×)	0.61 (6.51×)	0.67 (6.03×)
	16	0.14 (7.29×)	0.38 (8.95×)	1.50 (2.51×)	1.28 (3.05×)	0.94 (4.19×)	0.73 (5.38×)	0.61 (6.51×)	0.63 (6.41×)
refine	1	11.70 (1.00×)	14.73 (1.00×)	13.63 (1.00×)	13.67 (1.00×)	13.47 (1.00×)	13.73 (1.00×)	13.70 (1.00×)	13.73 (1.00×)
	2	7.36 (1.59×)	9.32 (1.58×)	9.38 (1.45×)	9.11 (1.50×)	9.19 (1.47×)	9.32 (1.47×)	9.13 (1.50×)	9.17 (1.50×)
	4	4.40 (2.66×)	5.53 (2.66×)	5.60 (2.43×)	5.49 (2.49×)	5.35 (2.52×)	5.36 (2.56×)	5.26 (2.60×)	5.34 (2.57×)
	8	3.15 (3.71×)	3.87 (3.81×)	4.29 (3.18×)	3.99 (3.43×)	3.90 (3.45×)	3.77 (3.64×)	3.74 (3.66×)	3.66 (3.75×)
	12	2.73 (4.29×)	3.32 (4.44×)	3.91 (3.49×)	3.62 (3.78×)	3.44 (3.92×)	3.36 (4.09×)	3.36 (4.08×)	3.31 (4.15×)
	16	2.45 (4.78×)	3.04 (4.86×)	3.61 (3.78×)	3.39 (4.03×)	3.21 (4.20×)	3.04 (4.52×)	3.00 (4.57×)	2.96 (4.64×)

Table 4. Execution times on $P = 1, 2, 4, 8, 12, 16$, in seconds, and their scalability profile for all benchmarks. Each of the replay columns shows the replay time with $P_{rep} = P$ workers replaying the same recorded execution (with $P_{rec} = P'$, as shown in the column heading). The numbers in the parenthesis indicate the speedup comparing to its single-worker execution counterpart, which has the 1.00 speedup. The highlighted cells indicate replay runs that uses the same number of workers as in the recording.

Recall the time bound for replay: its expected execution time on P workers is $O(W/P + S' \lg P)$, where W is the overall work in the computation and the S' is the span in the augmented DAG. Since $S \leq S' \leq S + B$, replay should scale as long as record scales if we ignore the $\lg P$ term. For most benchmarks, we do see that the replay scales similarly to the recorded execution when $P_{rec} = P_{rep}$ (the highlighted cells in Table 4), indicating that it is generally safe to ignore the $\lg P$ term and that the overheads of suspending and restarting in replay is small.

The two exceptions are data points in `matching` and `MIS`. There are two possible explanations. The first is that the augmented DAG is running out parallelism. We don't believe that this is the case, since if replay uses more workers $P_{rep} > P_{rec}$, we continue to see the execution scaling (i.e., by looking at the scalability of data points below the highlighted cells). The more likely explanation is the following: these benchmarks are already memory bound, and replay has a much larger memory footprint than record, causing additional cache misses, and the higher memory latency slows down the parallel execution. Indeed, these executions incur higher cache misses during replay than during record. There are two reasons for these additional cache misses. First, during replay, workers suspend their current deque from time to time (discussed in Section 3) and thus can create large number of suspended deques. Second, while record can use a Bloom filter and

do without a hash table, replay must use either an array or a hash table. While arrays have fewer cache misses than the hash list, both of these have a larger memory footprint than the Bloom filter. The overhead of replay was much higher when we used a hash table in our initial implementation, which requires even more memory than the array.

The additional memory footprint also in part explains the data points with higher standard deviation. The number of additional deques created during replay is a function of scheduling, and the number of suspended deques can differ from run to run. Execution times for benchmarks that are already memory bound will be more sensitive to this changes in the number of suspended deques.

6 COMPARISON BETWEEN PORRIDGE AND PARROT

To evaluate the benefit of processor-oblivious record and replay, we have developed a processor-aware record-and-replay system designed for dynamic multithreaded computations, called PARRot. The design of PARRot is specifically tailored to target a work-stealing runtime system — PARRot only records memory and lock accesses that can lead to nondeterministic scheduling choices, and enforces those choices during replay. Moreover, PARRot shares the same well-optimized data structures used by PORRidge for recording and replaying happened-before edges among worker threads, thereby allowing a fair comparison.

This section describes the high-level design of PARRot, empirically evaluates its overhead, and compares that against the overhead of PORRidge. The empirical results show that PORRidge not only consistently outperforms PARRot, it also obtains similar speedups as the baseline code (i.e., without record and replay), where PARRot fails to obtain similar speedups in some cases. Finally, PARRot failed to record and replay `chess` (described in Section 5) on the hardware platform used for evaluation due to out-of-memory errors, while PORRidge successfully records and replays it while exhibiting similar speedups observed in the baseline code.

6.1 PARRot: A Processor-Aware Record-and-Replay System

Why develop our own processor-aware record-and-replay system. We could have employed a generic record-and-replay system designed for multithreaded C code. In fact, we have tried using PinPlay [60], a general record-and-replay system based on Pin [51], a popular dynamic binary instrumentation framework. We used PinPlay to record and replay Delaunay Refinement (`refine`, described in Section 5) and found that it has 96.8× overhead for recording and 16.1× overhead for replay when executing the computation on one worker — 1-2 orders of magnitude worse than the PORRidge overheads of 1.26× and 1.16×, respectively. When we tried recording and replaying on multiple workers, the executions with PinPlay slowed down and showed no speedup.

Comparing PORRidge with an out-of-the-box record-and-replay system such as PinPlay is not fair, however, since PinPlay must capture all *potential* sources of non-determinism, including all results from system calls and every memory access to nonlocal variables (since they could be shared) even at the application level. PORRidge does not do this since it targets data-race free (DRF) Cilk programs. Moreover, PinPlay is based on a binary instrumentation framework, which is commonly thought to incur higher overhead than directly embedding instrumentation in the compiled code [12, 68, 73], as PORRidge does. Finally, the back-end data structures used by PinPlay and PORRidge to enforce thread interleavings may differ greatly, making it challenging to perform an apples-to-apples comparison.

The design and implementation of PARRot. We developed PARRot, our own version of a processor-aware record-and-replay system, to enable as fair a comparison to PORRidge as possible.

First, PARRot instruments the same set of events as PORRidge at the user application level. Note that even though PARRot records the scheduler's non-determinism and enforces the same

scheduling decisions during replay, it must still record and enforce the same happened-before relations due to lock acquisitions and releases performed in the application code. This is because the order in which tasks enter critical sections is not visible to the scheduler, and hence is not captured by recording the scheduler's non-determinism.

Second, PARRot uses most of the same back-end infrastructure as PORRidge. We have implemented PARRot based on the same Cilk Plus runtime system that PORRidge is based on. PARRot uses the same well-optimized data structures used by PORRidge to enforce thread interleavings (but does so at the scheduler level).

Finally, PARRot is specifically designed for dynamic multithreaded computations scheduled using work stealing. Even though many memory accesses to shared data structures are performed within the scheduling code, recording every single access is unnecessary as long as we enforce the same lock orderings used to protect accesses to these data structures. Thus, instead of recording every access to shared variables (which is what a generic record-and-replay system would do), we only record and enforce during replay a small subset of memory accesses, system calls, and lock acquisitions / releases based on our knowledge of the work-stealing scheduler.

There are potentially racy memory accesses in the scheduling code, where the results of the race, while not affecting correctness, could impact scheduling decisions.[6] In PARRot, such racy memory accesses are protected by fine-grained locks. The inter-thread interactions induced by these racy memory accesses are recorded via lock-acquisition ordering and enforced by replaying the same orderings.

6.2 Empirical Evaluation of PARRot and Comparison to PORRidge

We have empirically evaluated PARRot using the same set of benchmarks and the same hardware platform described in Section 5. Table 5 shows the execution characteristics of the scheduler, i.e., the number of locks and lock acquisitions performed by the scheduling code for each application when executing on different number of workers. For ease of comparison, the last row also shows the number of locks and lock acquisitions performed by the application code when running on one worker (multiple-worker runs can differ slightly but have similar values).

	number of locks						*total number of lock acquisitions*					
config	chess	dedup	ferret	matching	MIS	refine	chess	dedup	ferret	matching	MIS	refine
rts, 1P	8	8	8	25	20	4.8e3	1.6e9	4.6e5	1.2e3	3.1e6	3.0e6	1.3e6
rts, 2P	45	2.8e5	182	114	102	1.3e4	1.6e9	2.0e6	1.2e5	3.2e6	3.1e6	4.0e6
rts, 4P	218	4.0e5	261	406	362	2.5e4	1.6e9	3.0e6	2.3e5	3.2e6	3.1e6	1.0e7
rts, 8P	696	4.6e5	305	1426	1089	5.5e4	1.6e9	6.4e6	7.6e5	3.2e6	3.1e6	2.0e7
rts, 12P	1040	4.9e5	330	2170	1817	8.7e4	1.6e9	9.9e6	1.4e6	3.3e6	3.3e6	2.9e7
rts, 16P	1712	5.0e5	354	3162	2516	1.0e5	1.6e9	1.4e7	2.1e6	3.3e6	3.2e6	4.1e7
user code	4	1	1	5e6	5e6	4.8e7	2.8e4	7.3e5	256	5.0e7	2.8e6	1.2e7

Table 5. The execution characteristics of the scheduling code for each application. The rows with *rts* configuration show the the number of locks and lock acquisitions performed by the scheduling code for each of $P = 1, 2, 4, 8, 12, 16$ processors in addition to the lock acquisitions done as part of the application code. The last row (*user code*) shows the number of locks and lock acquisitions performed by the application code when running on one processor (same values as shown in Section 5 Table 1).

As we increase the number of processors used, the number of locks and lock acquisitions within the scheduling code increases, which is to be expected. As more workers are used, the scheduler dynamically creates more bookkeeping data used to keep track of actual parallel execution, and thus more lock acquisitions are performed to synchronize accesses to them.

[6] An example of such racy accesses include the "THE" protocol [34] commonly used by a work-stealing scheduler to coordinate accesses to a deque between a victim and a thief.

One can compare the relative numbers of locks and lock acquisitions between the scheduling code and the user code, which corresponds to the changes to the overhead of PARRot, which we discuss next in the context of overhead evaluation.

			PARRot		*PORRidge*	
application	*P*	*baseline*	*record*	*replay*	*record*	*replay*
chess	1	64.49	n/a	n/a	64.36 (1.00×)	65.14 (1.01×)
	2	32.20	n/a	n/a	32.20 (1.00×)	32.60 (1.01×)
	4	16.11	n/a	n/a	16.11 (1.00×)	16.35 (1.01×)
	8	8.15	n/a	n/a	8.13 (1.00×)	8.45 (1.04×)
	12	5.38	n/a	n/a	5.38 (1.00×)	5.66 (1.05×)
	16	4.04	n/a	n/a	4.11 (1.02×)	4.35 (1.08×)
dedup	1	48.04	49.14 (1.02×)	76.60 (1.59×)	48.20 (1.00×)	48.16 (1.00×)
	2	24.43	25.09 (1.03×)	46.94 (1.92×)	24.44 (1.00×)	24.44 (1.00×)
	4	12.43	12.91 (1.04×)	34.16 (2.75×)	12.40 (1.00×)	12.51 (1.01×)
	8	6.43	6.91 (1.07×)	29.74 (4.63×)	6.49 (1.01×)	6.46 (1.00×)
	12	4.52	5.03 (1.11×)	29.75 (6.58×)	4.53 (1.00×)	4.55 (1.01×)
	16	3.61	4.22 (1.17×)	32.10 (8.89×)	3.65 (1.01×)	3.64 (1.01×)
ferret	1	8.92	9.26 (1.04×)	9.29 (1.04×)	8.89 (1.00×)	9.10 (1.00×)
	2	4.52	4.75 (1.05×)	4.82 (1.07×)	4.57 (1.01×)	4.54 (1.00×)
	4	2.31	2.42 (1.05×)	2.66 (1.15×)	2.33 (1.01×)	2.32 (1.00×)
	8	1.27	1.30 (1.02×)	1.80 (1.42×)	1.24 (0.98×)	1.26 (0.99×)
	12	0.91	0.95 (1.04×)	1.94 (2.13×)	0.91 (1.00×)	0.92 (1.01×)
	16	0.76	0.79 (1.04×)	1.90 (2.50×)	0.78 (1.03×)	0.74 (0.97×)
matching	1	3.06	11.80 (3.86×)	13.07 (4.27×)	9.64 (3.15×)	10.07 (3.29×)
	2	1.92	7.17 (3.73×)	11.01 (5.73×)	5.78 (3.01×)	6.82 (3.55×)
	4	0.96	3.82 (3.98×)	5.73 (5.97×)	3.03 (3.16×)	3.75 (3.91×)
	8	0.50	2.34 (4.68×)	3.18 (6.36×)	1.68 (3.36×)	3.22 (4.44×)
	12	0.32	1.45 (4.53×)	2.03 (6.34×)	1.13 (3.53×)	1.77 (5.53×)
	16	0.25	0.99 (3.96×)	1.75 (7.00×)	0.89 (3.56×)	1.57 (6.28×)
MIS	1	1.02	4.92 (4.21×)	4.49 (4.40×)	3.40 (3.33×)	3.77 (3.70×)
	2	0.65	2.61 (4.02×)	4.07 (6.26×)	2.03 (3.12×)	2.54 (3.91×)
	4	0.32	1.56 (4.88×)	2.17 (6.78×)	1.03 (3.22×)	1.36 (4.25×)
	8	0.16	0.74 (4.63×)	1.23 (7.69×)	0.58 (3.63×)	0.79 (4.94×)
	12	0.13	0.48 (3.69×)	1.03 (7.92×)	0.42 (3.23×)	0.61 (4.69×)
	16	0.14	0.40 (2.86×)	1.04 (7.43×)	0.38 (2.71×)	0.63 (4.50×)
refine	1	11.70	19.17 (1.64×)	17.87 (1.53×)	14.73 (1.26×)	13.63 (1.16×)
	2	7.36	13.13 (1.78×)	14.87 (2.02×)	9.32 (1.27×)	9.11 (1.24×)
	4	4.40	8.29 (1.88×)	11.27 (2.56×)	5.53 (1.26×)	5.35 (1.22×)
	8	3.15	5.29 (1.68×)	9.92 (3.15×)	3.87 (1.23×)	3.77 (1.20×)
	12	2.73	4.78 (1.75×)	9.96 (3.65×)	3.32 (1.22×)	3.36 (1.23×)
	16	2.45	4.64 (1.89×)	12.53 (5.11×)	3.04 (1.22×)	2.96 (1.21×)

Table 6. Execution times in seconds on $P = 1, 2, 4, 8, 12, 16$ and their overheads comparing to the baseline for all benchmarks. The *baseline* column shows the running times without any record replay using the original Cilk Plus runtime. The *PARRot* columns show the running times for recording and replaying using the conventional strategy that records all non-determinism within runtime and enforcing the same scheduling decision in the runtime code during replay. The *PORRidge* columns show the running time for performing record and replay using the PORRidge strategy. For both strategies, the same number of workers are used for record and replay runs. The cells showing N/A indicate that the system failed to record / replay due to out-of-memory errors.

Table 6 shows the execution times of recording and replaying using PARRot on $P = 1, 2, 4, 8, 12, 16$ cores and their overhead (in parentheses) compared to the baseline execution (i.e., without record and replay). The numbers for PORRidge are shown as well (the same numbers as shown in Section 5) for ease of comparison.

PORRidge outperforms PARRot for every configuration (i.e., across different applications and number of workers used). Two factors contribute to this. First, PARRot must incur additional overhead for recording non-determinism in the scheduling code. Second, with PARRot, the replay may cause the workers to perform additional blocking in order to enforce the same scheduling decisions, whereas PORRidge is free to perform dynamic load balancing depending on how the scheduling plays out. Due to this blocking, PARRot generally incurs higher overhead during replay than during record, whereas for PORRidge the opposite is true.

The amount of additional overhead incurred by PARRot during recording, compared to that of PORRidge, is not uniform across benchmarks, and can be explained by the execution characteristics shown in Table 5. The higher the number of lock acquisitions performed in the scheduling code relative to that in the application code, the higher the overhead PARRot incurs during recording. For instance, the relative overhead incurred by PARRot during recording compared to that of PORRidge is the smallest for `MIS`, which also has the smallest increase in lock acquisitions in the scheduling code relative to the lock acquisitions already performed by the application code.

The amount of additional overhead incurred by PARRot during replay, compared to that of PORRidge, can be attributed to blocking due to the fact that certain scheduling decisions must be enforced. Since the scheduler in PARRot has to exactly replicate the steal patterns of the recording, it may cause a worker thread to spin-wait (instead of doing useful work) when it reaches a recorded inter-thread interaction before the other thread gets there. Note that such inter-thread interactions include all failed steal attempts between a thief and a victim worker, since a failed steal attempt is communicated through shared memory accesses. In contrast, PORRidge never spins during replay and uses suspension to explore all possible parallelism in the augmented DAG.

Finally, PARRot failed to record (and thus replay) `chess` due to out of memory errors; the process quit when it reached 64 GByte memory usage, which is the size of the DRAM on the evaluation platform. As shown in Table 5, `chess` has a large number of lock acquisitions in the scheduling code, most of which are caused by spawn and sync statements that induce potentially racy memory accesses to the per-worker deque that keeps track of available work. In the baseline execution, memory accesses to a worker's deque may not always require synchronization; specifically the "THE protocol" [34] employed by the Cilk Plus runtime uses a Dekker-like protocol and causes the victim to acquire the lock on its own deque only if it is likely to conflict with a thief stealing. These memory accesses are potentially racy and can impact scheduling decisions, however. Therefore, such shared memory accesses must be recorded and replayed faithfully when using a processor-aware record-and-replay strategy. Consequently, the recording using PARRot produced an overwhelmingly large log that eventually caused the system to run out of memory. In contrast, the processor-oblivious strategy used by PORRidge is not affected by this; it was able to record and replay `chess` with little overhead while obtaining near-linear speedups.

6.3 Discussion: Benefits of Processor Obliviousness

As our experimental data indicates, processor-oblivious record and replay can be implemented efficiently. The only time PORRidge exhibits non-negligible overhead is when the benchmark is already memory bound (i.e., `matching` and `MIS`). Nevertheless, for such cases, a processor-aware record-and-replay system such as PARRot needs to log additional information to record the inherent non-determinism in the scheduler, which further increases the memory footprint and overhead of the recording, as the data in Table 6 shows.

For replay, PARRot can incur high overhead due to the fact that it must exactly replicate the steal patterns of the recording, potentially blocking and causing more idleness. If a recording is done on P workers and takes x time, in a processor-aware system such as PARRot, the replay cannot run in less than x time (asymptotically) no matter how many workers we give it. As the data shows, the running time tends to be slower. In contrast, PORRidge never spins during replay and uses suspension to explore all the possible parallelism in the augmented DAG. These results in part speak to the performance advantage of PORRidge's approach, because PORRidge does not need to reproduce the runtime's non-determinism while a processor-aware system such as PARRot must.

Moreover, the strategy used by PORRidge has the additional benefit of scaling the replay beyond the number workers used in record, which a processor-aware record-and-replay system such as PARRot cannot provide. Therefore, as the experiments indicate, replay often runs just as fast as the record when $P_{rep} = P_{rec}$, and can continue to scale when $P_{rep} > P_{rec}$ (data shown in Section 5).

7 RELATED WORK

Record and Replay. To our knowledge, all software-based record-and-replay systems are tied to thread-based programming models: a runtime system records the behavior and interleaving of the threads in the program, and on replay re-runs the same threads with the same behavior. Recording and replaying on the same number of threads simplifies both the recording process (as thread-based identifiers can be used to identify operations) and the replay process (as there is no need to map operations from the recorded run onto a different number of threads). RecPlay [67] and JaRec [35] do not handle racy accesses, and have reasonable overhead, but, as with PORRidge, are unsound in the presence of races.

Racy accesses are more challenging, since accesses to shared memory result in happened-before edges that must be preserved during replay. For systems that handle racy accesses, there are several approaches. Some speculate that races are infrequent or irrelevant to keep recording overhead down [45, 75]. Some preserve a limited amount of information during record and rely on offline search or constraint-solving approaches to generate the information required for replay [3, 38, 49, 59]. Some systems track racy interleavings directly, which either add large overhead [43, 78], use coarse-granularity communication tracking (such as page-based conflict detection) that can be overly-conservative [30, 42], or rely on carefully modified virtual machines [21].

One could apply a traditional thread-based record-and-replay system on dynamic multithreaded computation directly, and record all sources of nondeterminism in order to replay deterministically. PinPlay [60] is such a general record-and-replay system based on Pin, a popular dynamic binary instrumentation framework [51], that captures all sources of nondeterminism including racy memory accesses, thread interleavings, and results from system calls. Using such a general purpose record-and-replay system on dynamic multithreaded computations can incur substantial overhead, however, as we discussed in Section 6.

Chimera [44], another record-and-replay system for pthreaded code, on the other hand, uses static race detection to identify potentially-racing pairs of accesses, and uses lightweight synchronization, as well as lock coarsening, to enable a simple record and replay technique. Such an approach, could be adapted to make PORRidge applicable to racy Cilk programs.

Another strategy is to record information at the *hardware* level [37, 56, 61, 76], by piggy-backing on cache-coherence protocols to record communication between different hardware contexts. While these systems could, in principle, be used to record the behavior of Cilk programs and to capture the non-determinism introduced by the scheduler, they have two drawbacks: 1) like existing software-based models, their (hardware) context-based recording system constrains replay to run with the same level of parallelism as record; 2) they require hardware modifications, and hence do not work in any existing commodity systems.

Determinism. A related technique is *deterministic execution*, where a combination of programming model constraints and runtime checks ensures that an application always produces the same behavior when presented with the same input. Note that this is subtly different than record and replay: in record and replay, different *recorded* runs can exhibit different behaviors; replay must replicate whichever recorded run it is replaying. One approach to determinism is to mandate it through programming model restrictions [11, 15, 20, 23, 57, 65] which generally preclude general use of locks and other synchronization mechanisms. Moreover, while some of these approaches can provide determinism independent of the number of threads [15, 57], most do not. Another approach is to enforce determinism through hardware [27, 28], compiler [9], OS [5, 10] or runtime approaches [50, 58]. While these techniques do not require specialized programming models, these techniques are usually not processor oblivious.

Dynamic Analyses for Dynamic Multithreading. The most common analysis tool for dynamic multithreading programming models is *on the fly race detection* — for a given input, these tools run the program on that input *once* while keeping track of enough information that allows them to report a race if and only if the program contains a race on that input. Over the years, researchers have proposed algorithms for doing this both sequentially [1, 31, 63] and in parallel [8, 53, 64, 74, 77]. Some of these have led to implementations [31, 64, 74, 77]. The parallel tools are generally processor oblivious; a single run on any number of workers gives the correct answer. Another important class of tools is *performance profilers*, that either measure work and span of the program directly during execution [36, 69] or use sampling to determine where in the code causes workers to be idle [72].

Work-stealing Runtime with Multiple Deques. Prior work-stealing designs have used more than P deques for supporting concurrent data structures [2, 74], blocking I/O operations [55, 79], or the use of futures [71]; some provide theoretical scheduling bounds [2, 55, 71, 74], but their modifications are for a different purpose and require different modifications and analyses.

8 CONCLUSIONS

This paper presented the first processor-oblivious record and replay scheme for data race-free dynamic multithreaded programs. This scheme is provably good, efficient in practice, and provides good scalability.

There are many directions of future work. First, we could target a richer set of primitives that induce happened-before relationships; for instance, `try-lock` and `compare-and-swap`. These require rethinking the exact semantics we want from a happened-before edge, since, in some cases, programs use the non-determinism induced by these mechanisms to enable efficiency, complicating which edges we want to record. Second, we could try to expand to programs with data races — this would involve recording happened-before relationships not just between critical sections, but also between accesses to memory locations that could be involved in races. As mentioned in the introduction, this does not require conceptual changes to PORRidge, just the ability to indicate to PORRidge where non-determinism due to races might occur. Finally, we can explore other mechanisms to enable processor-oblivious record and replay to see if some of them will give better performance.

PORRidge is open source and currently available at https://github.com/wustl-pctg/porridge. The library is provided under The MIT License, while the runtime modifications are licensed separately under a BSD license. The repository contains complete instructions for compiling and using PORRidge, in addition to scripts that reproduce most of the empirical results. Please send feedback or file issues at our github repository to help us continually improve the project.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation under grant numbers CCF-1150036, CCF-1218017, XPS-1439062, CCF-1150013, CCF-1439126, CCF-1527692, and CCF-1733873, and Department of Energy under grant number DE-SC0010295. We thank the referees for their excellent comments.

REFERENCES

[1] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. 2018. Race Detection and Reachability in Nearly Series-Parallel DAGs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM, New Orleans, Louisiana, 156–171.

[2] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably Good Scheduling for Parallel Programs That Use Data Structures Through Implicit Batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. ACM, Prague, Czech Republic, 84–95.

[3] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles (SOSP '09)*. ACM, Big Sky, Montana, USA, 193–206.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*. Puerto Vallarta, Mexico, 119–129.

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-Enforced Deterministic Parallelism. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC, Canada, 1–16.

[6] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (March 2009), 404–418.

[7] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Orlando, Florida, USA, 735–736.

[8] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010a. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, Pittsburgh, Pennsylvania, USA, 53–64.

[10] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010b. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Vancouver, BC, Canada, 177–191.

[11] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Orlando, Florida, USA, 81–96.

[12] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE '11)*. ACM, Szeged, Hungary, 9–16.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[14] Christian Bienia and Kai Li. 2010. Characteristics of Workloads Using the Pipeline Programming Model. In *Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture*. Springer Berlin Heidelberg, 161–171.

[15] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proceedings of Principles and Practice of Parallel Programming*. ACM, 181–192.

[16] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

[17] Robert D. Blumofe. 1995. *Executing Multithreaded Programs Efficiently*. Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55–69.

[19] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.
[20] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA, USA, 4–9.
[21] Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. 2015. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *ACM International Conference on Principles and Practice of Programming in Java*. 90–101.
[22] Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective* (3rd ed.). Pearson, USA.
[23] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Reno/Tahoe, Nevada, USA, 691–707.
[24] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61.
[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 519–538.
[26] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*.
[27] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. Washington, DC, USA, 85–96.
[28] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A relaxed consistency deterministic computer. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 67–78.
[29] E. W. Dijkstra. 1968. Co-operating Sequential Processes. In *Programming Languages*, F. Genuys (Ed.). Academic Press, London, England, 43–112. Originally published as Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
[30] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. 2008. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*. ACM, Seattle, WA, USA, 121–130.
[31] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. ACM.
[32] Jeremy T. Fineman and Charles E. Leiserson. 2011. Race Detectors for Cilk and Cilk++ Programs. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1706–1719.
[33] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 79–90.
[34] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 212–223.
[35] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. 2004. JaRec: A Portable Record/Replay Environment for Multi-threaded Java Applications. *Software Practice & Experience* 34, 6 (May 2004), 523–547.
[36] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. Santorini, Greece, 145–156.
[37] Derek R. Hower, Pablo Montesinos, Luis Ceze, Mark D. Hill, and Josep Torrellas. 2009. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *Commun. ACM* 52 (June 2009), 93–100. Issue 6.
[38] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *ACM Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, Seattle, Washington, USA, 141–152.
[39] Institute of Electrical and Electronic Engineers. 1997. Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]. IEEE Standard 1003.1, 1996 Edition. (1997).
[40] Intel Corporation 2009. *Intel(R) Threading Building Blocks*. Intel Corporation. Available from http://www.threadingbuildingblocks.org/documentation.php.

[41] Intel Corporation 2010. *Intel Cilk Plus Language Specification.* Intel Corporation. Document Number: 324396-001US. Available from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf.

[42] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*. ACM, New York, New York, USA, 155–166.

[43] T. J. LeBlanc and J. M. Mellor-Crummey. 1987. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.* 36 (April 1987), 471–482. Issue 4.

[44] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. Chimera: Hybrid Program Analysis for Determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, Beijing, China, 463–474.

[45] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, Pittsburgh, Pennsylvania, USA, 77–90.

[46] I-Ting Angelina Lee and Tao B. Schardl. 2015. Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA'15) (SPAA '15)*. ACM, Portland, OR, USA, 111–122.

[47] Daan Leijen and Judd Hall. 2007. Optimize Managed Code For Multi-Core Machines. *MSDN Magazine* (2007). Available from http://msdn.microsoft.com/magazine/.

[48] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. 2012. Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*.

[49] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. 2015. Light: Replay via Tightly Bounded Recording. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, Portland, OR, USA, 55–64.

[50] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Cascais, Portugal, 327–336.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*. Chicago, IL, USA, 190–200.

[52] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, Xi'an, China, 693–708.

[53] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. Albuquerque, NM, USA, 24–33.

[54] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. 2000. The Power of Two Random Choices: A Survey of Techniques and Results. In *in Handbook of Randomized Computing*. Kluwer, 255–312.

[55] Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Pacific Grove, California, USA, 71–82.

[56] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. 2006. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, San Jose, California, USA, 229–240.

[57] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, Portable and Parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, Salt Lake City, Utah, USA, 499–512.

[58] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, Washington, DC, USA, 97–108.

[59] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, Big Sky, Montana, USA, 177–192.

[60] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, Toronto, Ontario, Canada, 2–11.

[61] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai. 2009. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, New York, 576–585.

[62] Martin Raab and Angelika Steger. 1998. "Balls into Bins" — A Simple and Tight Analysis. In *Randomization and Approximation Techniques in Computer Science*, Michael Luby, José D. P. Rolim, and Maria Serna (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–170.

[63] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

[64] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

[65] Martin C. Rinard and Monica S. Lam. 1998. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems* 20 (1998), 483–545. Issue 3.

[66] Arch D. Robison. 2013. Cilk Plus Solver for a Chess Puzzle or: How I Learned to Love Fast Rejection. https://software.intel.com/en-us/articles/cilk-plus-solver-for-a-chess-puzzle-or-how-i-learned-to-love-rejection. (Feb. 2013).

[67] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems* 17 (1999), 133–152. Issue 2.

[68] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 2, Article 43 (Dec. 2017), 25 pages.

[69] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA '15) (SPAA '15)*. Portland, Oregon, USA, 89–100.

[70] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the Problem Based Benchmark Suite. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*.

[71] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, Washington, District of Columbia, 257–271.

[72] Nathan R. Tallent and John M. Mellor-Crummey. 2009. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, Raleigh, NC, USA, 229–240.

[73] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. 2006. Analyzing Dynamic Binary Instrumentation Overhead. In *WBIA Workshop at ASPLOS, 2006*.

[74] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Pacific Grove, California, USA, 83–94.

[75] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Newport Beach, California, USA, 15–26.

[76] Min Xu, Rastislav Bodik, and Mark D. Hill. 2003. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *ACM/IEEE International Symposium on Computer Architecture*. ACM, San Diego, California, 122–135.

[77] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. 2018. Efficient Parallel Determinacy Race Detection for Two-dimensional Dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, Vienna, Austria, 368–380.

[78] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. 2011. ORDER: Object Centric Deterministic Replay for Java. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Portland, OR, 30–30.

[79] Christopher S. Zakian, Timothy A. K. Zakian, Abhishek Kulkarni, Buddhika Chamith, and Ryan R. Newton. 2016. *Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++*. Springer International Publishing, Cham, 73–90.

Received August 2018; revised April 2019; accepted May 2019