Processor-Oblivious Record and Replay

Robert Utterback Kunal Agrawal I-Ting Angelina Lee Milind Kulkarni

Washington University in St. Louis
{robert.utterback,kunal,angelee}@wustl.edu

Purdue University
milind@purdue.edu

Abstract

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 such 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.

Categories and Subject Descriptors F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems—Sequencing and scheduling; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Tracing

Keywords deterministic replay; dynamic program analysis; reproducible debugging; work stealing

PPoPP '17, February 04-08, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3018743.3018764

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*** [2, 20, 29, 34, 36, 37, 41–43, 48, 53, 56, 58, 63, 68–70]. 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 particular, 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 record run and the replay run. Prior work on record and replay generally records *thread* interleaving, and tracking the behavior of the threads of a program as they execute, ensuring that during replay, threads interleave in the same way when executing critical sections. While this approach succeeds at its goal of replaying recorded behavior, it has the drawback of requiring that the replay run use the same number of threads as the recorded execution.

This concession seems mild: most programming models make the number of threads an explicit parameter. 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, 32, 40], subsets of OpenMP [6], Threading Building Blocks [39], the Habanero family [8, 23], Task Parallel Library [46], X10 [23, 24], 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 responsi-

ble for efficiently mapping this parallelism to ***worker*** threads that execute the computation in parallel.[1]

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 exist no record and replay systems for dynamically multithreaded programming models. Even if we keep the number of workers (threads) the same for recording and replaying, standard record and replay mechanisms do not directly work due to the Cilk scheduler's use of ***randomized work stealing***: *which* workers execute *which* tasks *when* is also non-deterministic and can change from one execution of a computation to the next even if the number of workers remains the same. If we record these computations using a standard record and replay scheme, then it would have record all of the decisions of the scheduler and then reproduce these exact decisions during replay significantly increasing cost of both record and replay.

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.[2] We also assume that there are no parallelism within critical sections, which is a standard assumptions 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.

[1] We use workers and processors interchangeably in this paper.

[2] 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 [34, 63], as well as other dynamic analyses [55]. 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., [25, 30, 31, 45]) to identify races and indicate to PORRidge additional sources of non-determinism (tools like Chimera adopt similar approaches [44]).

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 needs 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, 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 acquires represent additional ***happens-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 happens-before edges. More importantly, during the replay run, PORRidge ensures that the happens-before relationships that were recorded are respected: in other words, during replay, PORRidge schedules the ***augmented DAG*** which contains all these happens-before edges in addition to the original dependencies. While this new augmented DAG may have parallelism limited by the happens-before edges, its parallelism is *not* directly limited by the number of threads that the recording run executed on.

Another important property of PORRidge is that the recording system sits *on top* of its runtime. One possible way to record a Cilk computation is to include the Cilk runtime in the scope of what is recorded, recording and subsequently replaying all of the non-deterministic decisions regarding work stealing. But the Cilk runtime is highly parallel and non-deterministic, and including it in the recording scope would dramatically increase the amount of information to be recorded. Instead, PORRidge only records the happens-before relationships.

Replay is more complex. the Cilk runtime system is not designed to obey happens-before edges that are not directly part of the program itself. Therefore, PORRidge adds mechanisms to the Cilk runtime system to respect these dependencies. 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 happens-before relationships between critical sections. 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. Despite the fact that PORRidge requires additional happens-before tracking during record, and requires conforming to those happens-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 happens-before edges, its runtime is $O(W/P' + S' \log\log P')$, where S' is the span of the augmented DAG (i.e., with the additional happens-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 implemented a prototype of our design within the Cilk Plus [40] runtime system. We show across six benchmarks that PORRidge delivers 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 happens-before edges, the additional overhead of replay over the record is small.

2. PRELIMINARIES

We now provide background on modeling parallel computations, work-stealing schedulers, and some definitions.

Dynamic Multithreading and Computation DAGs. Here we will use Cilk Plus programming keywords, `cilk_spawn` and `cilk_sync`, to explain the dynamic multithreaded programming model; other languages may differ in keywords. 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. Instruction `cilk_sync` 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, where each node is a unit time computation and each edge represents a dependence between nodes — a particular node is ***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 [17, 32] dynamically load balances a parallel computation across available ***worker*** threads. Each worker maintains a ***deque***, double-ended queue, to keep track of available work. When a worker creates new strands, they are placed on this worker's deque. When it completes its current strand, it takes work from the bottom of the deque. If its deque becomes empty, the worker turns into a ***thief*** and chooses a ***victim*** worker 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 [17].

Modeling Critical Sections. Since our computations contain critical sections, we must model those. We assume that 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 a *acquire* node and the last node is called a *release* node. We say B_1 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 happens-before edges are added to the computation DAG. We call the new DAG with the happens-before edges an ***augmented dag***. More precisely, if critical section b accesses lock ℓ_i after 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 (acquire 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 $\tilde{S}$.

[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. DESIGN OF PORRidge

We now describe the design and implementation of PORRidge. As mentioned in Section 1, since the PORRidge is designed for data-race free programs, it needs to capture only the happens-before edges formed between critical sections protected by the same lock during recording and enforce the same happens-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.

3.1 Record

Conceptually, during recording, PORRidge stores with each lock an ordered list of successful lock acquires to this lock, henceforth referred to as the ***lock-acquire ordering***. Conceptually, a lock object in PORRidge contains a pointer to the underlying lock defined by the POSIX pthread specification [38] and a data structure recording its lock-acquire ordering. PORRidge provides wrappers for the various thread lock objects and associated acquire / release functions; during recording, the wrapper functions are invoked via compile-time interpositioning [21][Chp. 7.13] to record the necessary information. When a worker successfully acquires a lock, it simply adds the currently executing strand to the end of its lock-acquire ordering. If the lock is not available, the worker spins. At the end of the recorded execution, every lock object writes out the strands in its lock-acquire ordering to a log file in the order inserted.

Identifying Strands. For processor-oblivious replay, the information stored in the list must uniquely identify critical sections in the computation dag, and the identification must be consistent across executions. Here, we use ***pedigree*** [47], a sequence of integers corresponding to the rank ordering of spawn statements in the ancestor functions (including this function) that lead to the current strand. Pedigree uniquely identifies each strand in a consistent manner since it depends only on the computation dag and not on the schedule. Critical sections 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.

The open-source Cilk Plus runtime [40] 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 acquire, this causes lock acquires to incur $\Theta(d)$ overhead during record and replay. Ideally, we would like to keep the cost of lock acquire 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 [47] to give each critical section a ***section ID***, which is effectively a hash of a 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 the 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 [47]. Using a similar strategy as DOTMIX, we obtain unique section IDs with constant overhead per lock acquire.

Storing Strand IDs. The drawback of section IDs is the (rare) possibility of collision. We can detect these collisions by maintaining the lock-acquire ordering for each lock using a ***hash-list*** representation, which is a combination of hash table and list. The hash list implicitly maintains a linked list and records lock-acquire ordering using this linked-list, but in addition hashes each list node in a hash table using the corresponding section ID as the key so that the hash table disambiguates lock acquires with the same section IDs.

During record, when a new lock acquire occurs, its section ID gets inserted into the end of the implicit list. PORRidge then checks the section ID in the hash table; if there are no collisions (no entry with the same section ID exists in the hash table), entry for this acquire is simply inserted to the hash table. If there is a collision, it marks the first entry with the same section ID in the hash table as "has collision," reads the full pedigree of the current section, and stores it in the hash table using chaining. Therefore, each entry in the hash table has two "next" pointers: one for the implicit list ordering (points to the next section to acquire the lock) and one for chaining (points to the full pedigree of a section that has the same section ID).

3.2 Replay

At the beginning of the replay, the runtime reads in the previously recorded log and recreates the lock-acquire ordering represented using a hash list. PORRidge maintains invariant that the head of the implicit 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 acquire for critical section a, and its predecessor — a lock release of the critical section b that was executed immediately before a during the recording 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 b) releases a lock, it may enable critical section a (which was tried earlier and suspended); this worker must then resume this suspended critical section.

Lock Acquire. When a worker encounters a lock acquire, it checks the head of the list to see if this is the strand

that should get the lock next. If so, it acquires the lock and continues execution. Otherwise, the worker hashes its section ID and marks the corresponding hash list entry to indicate that the corresponding lock acquire 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 leads to bad performance, it can also lead to deadlocks. Consider an example where we record an execution on multiple workers and replay it 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 happens-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 gotten 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 can 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 acquire has been tried and suspended. If not, the worker simply continues the execution after the lock release. If the next lock acquire 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 acquire 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 acquire 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 acquire in line — the synchronization is coordinated using a Dekker-like protocol [28], since there are at most two workers concurrently operating on a given list node.

Handling Section-ID Collisions. When a worker encounters a lock acquire and tries to determine if this lock acquire is at the head of the lock-acquire ordering, it may find the head list node marked as"has collision," — in this case, multiple sections with the same ID acquired this lock. Recall that during recording, the runtime stores full pedigrees only when it discovers a section-ID collision; therefore, the first lock acquire involved in the collision has its section ID stored, and the remaining lock acquires involved in the collision have their full pedigrees stored. Thus, if a worker finds a head list node with the same section ID but marked as "has collision," it must read its full pedigree and compare it against other list nodes hashed with the same section IDs (for which the full pedigree is stored). Only if none of them match the current pedigree, the worker can conclude that it is at the head of the list and proceed with getting the lock; otherwise, it must suspend.

Runtime Modifications. The fact that a lock acquire 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 a worker can suspend execution 1) upon a lock acquire if the lock acquire is not ready, or 2) upon a lock release, if the lock release in turn enables a suspended lock acquire. 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 of the lock release, and resumes the deque containing the lock acquire 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 acquire is never on top of any deque and therefore no one ever steals it. When a worker suspends a deque due to a lock acquire that is not ready, the suspended lock acquire is at the bottom of the deque, and everything above it in the deque is ready. If the suspended deque contains only the lock acquire, the PORRidge runtime frees the deque. The suspended lock acquire, 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 acquire in s in the augmented dag. Therefore, if the lock acquire 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 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 w suspends a non-empty deque, it picks two workers uniformly at random and gives the deque to the worker with the smaller number of deques; and (2) on every steal attempt, the thief looks at two workers (chosen randomly), takes a suspended deque

from the worker with a larger number of deques and gives it to the worker with the smaller number of deques. 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. We shall see how these changes allow us to provide a provably-scalable replay time bound in Section 4.

3.3 Performance Optimization

Thus far we have been discussing the design assuming that the lock-acquire ordering for a given lock is represented using a hash list. The hash-list representation works, but it 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 list inherently lack locality and incur additional cache misses. We optimized the implementation of the record phase in PORRidge by using a small bloom filter to detect 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 list. 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 experiment, however, we find that using the bloom-filter outperforms the hash list due to cache effects.

Even though we were able to use a bloom filter during recording, the same optimization does not work during replay, since a worker uses the hash table not to identify collision but rather to find the list node corresponding to the encountered lock acquire quickly. During replay, a worker encountering a lock acquire that is not yet at the head of the list needs to find the corresponding list node in order to mark it suspended. If the corresponding section ID is marked to have collision during recording, the worker also needs to search through the list nodes with the same section ID to precisely identify the correct list node. A bloom filter is not sufficient for these purposes. Nevertheless, for many benchmarks, the number of lock acquires for a give lock is small; thus, keeping the lock-acquire ordering in a simple array and searching through the array suffices. For such benchmarks, the spatial locality and decrease in memory usage when using a simple array outweighs the benefit of constant-time search via a hash-list. Since the number of lock acquires per lock is known at the beginning of the replay, in our implementation, we optimized the replay to choose between the two representations — PORRidge keeps the lock-acquire ordering in a simple array if the number is small, and it uses a hash-list otherwise.

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 lock acquire, 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 will prove theoretical upper bounds on the running time of our record and replay strategy. Recording and replaying are done in different processes and we will 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 will extensively use the analysis of work stealing using a potential function provided by Arora, Blumofe and Plaxton [3] (abbreviated as ABP).

4.1 Running Time of Record

THEOREM 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 shall 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 2. *The total amount of time spent in blocking phases is at most B.*

Proof. Obvious from the fact that the total time any processor could be holding the lock is at most B.

LEMMA 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 [3], 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 2 and 3 gives us the stated theorem.

4.2 Running Time of Replay

THEOREM 4. *Given an augmented dag with work W and span $\tilde{S}$, the replay process completes in expected time $O(W/P + \tilde{S} \lg \lg P)$.*

As with the analysis of replay, we divide 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 will 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 $((\text{number of deques}) \times S)$ 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 5. *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 on the order of $O(P)$, 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.

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 of the 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 a phase is either work-bounded or steal-bounded.

LEMMA 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 5 gives the lemma.

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 APB to bound the steal attempts. At a very high level, the APB 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 $O(X)$ number of 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 1 plus the maximum depth of all its parents. The weight of a node is $w(u) = \tilde{S} - d(u)$. Then, we define a potential as follows:

DEFINITION 1. *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 APB analysis in a straightforward manner.

LEMMA 7. *The initial potential is $3^{2\tilde{S}-1}$ and it never increases during the computation.* □

The following lemma is a straightforward generalization of Lemmas 7 and 8 in ABP [3].

LEMMA 8. *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. We consider two scenarios. First scenario is where there are at most $2P$ suspended deques in the system, and we can bound the maximum load in this case.

LEMMA 9. *Say there are at most $2P$ suspended deques over all processors. With probability at least $1 - 1/P^2$, the processor with the largest load has at most $k = \lg\lg P + O(1)$ suspended deques.*

Proof. The lemma follows from the Azar et. al's [5, 7] classic balls into bins results. They prove that if we throw K balls into P bins by checking two bins and throwing the ball into the less loaded bin, then the maximum load is $\lg\lg P + O(K/P)$ with high probability. That is, the loads in bins are mostly balanced within an additive factor of $\lg\lg P$. 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 evenly. We will say that a distribution is ***balanced*** if the processor with the largest number of deques has fewer than $2\lg\lg P$ deques, otherwise, we will say that the distribution is unbalanced. We must also worry about imbalance creeping in as processors steal from suspended deques and suspended deques disappear. However, the proof of Theorem 4.1 from Azar et. al's paper [7] implies that if we start from an imbalanced distribution, and on each step, pick two random bins, and move a ball from the more loaded bin to the less loaded bin, then after $P^2 \lg\lg P$ steps, bins will be balanced again. Since our strategy for steals from Section 3 follows exactly this strategy, the following claim follows:

CLAIM 10. *If we start from a balanced distribution, it becomes imbalanced after $P^2 \lg\lg P$ steal attempts with probability at most $1/P^2$. If the distribution becomes imbalanced, it becomes balanced again after $P^2 \lg\lg P$ steal attempts with probability at least $(1 - 1/P^2)$.*

In the other scenario, where there are more than $2P$ suspended deques in the system, we cannot readily bound the maximum load, but one can show that such a scenario falls under the work-bounded phase with high probability:

LEMMA 11. *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 $\binom{P}{P/2}(1/2)^{2P} \leq (2e/16)^{P/2}$.

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

LEMMA 12. *Let $\Phi(t)$ denote the potential at the beginning of a good round. After $P \lg\lg P$ steal attempts, at the end of the round, the potential is at most $3\Phi(t)/4$ with probability at least $1/4$.* □

Proof. There are at most $2P \lg\lg P$ deques during the round. Therefore, the probability that a particular steal attempt hits a particular deque is at least $1/(2P \lg\lg P)$ (it may be higher since some workers have fewer than $\lg\lg P$ suspended deques). Therefore, we can apply a small modification to Lemma 8 generalized from ABP and argue that the total potential decreases.

LEMMA 13. *The total number of good rounds is $O(\tilde{S})$ in expectation.*

Proof. Similar arguments to ABP. At a high level, from Lemma 12, 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 $3^{2\tilde{S}}$. Therefore, after $O(\tilde{S})$ rounds, the potential disappears and the computation completes.

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

LEMMA 14. *The number of bad rounds is $O((1/P)X)$ where X is the number of good rounds.*

Proof. A round is good with probability at least $1 - 1/P^2$ from Lemma 9, Claim 10, and Lemma 11. If we ever get into a bad round, things become balanced again after $O(P^2 \lg\lg P)$ steal attempts, or $O(P)$ rounds from Claim 10. Therefore, it takes P^2 good rounds before a bad round occurs and then there can be at most P bad rounds before a good round occurs again. The following lemma follows from Lemmas 13 and 14 and the fact that each round has $P \lg\lg P$ steals.

LEMMA 15. *The expected number of steal attempts in steal bounded phases is at most $O(\tilde{S}P \lg\lg P)$.*

The following lemma follows from Lemmas 6 and 15

LEMMA 16. *The total number of steal attempts across all phases is $O(W + \tilde{S}P \lg\lg P)$.*

Lemma 16 and the facts that the total amount of work is W implies Theorem 4.

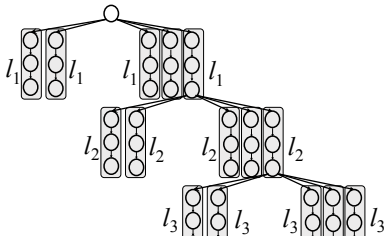

Figure 1: An example DAG with multiple locks where getting a tight bound for recording is impossible for an online scheduler. The offline scheduler can always schedule the "important" (in this case, the right-most) critical section first, but an online scheduler has no way of knowing which critical section 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 S, there is a bad dag where the next layer is always created by the critical section this strategy S executes last.

Let us now consider replay. In this case, W/P, and $\tilde{S}$ are lower bounds; therefore the replay bound of $O(W/P + \lg\lg P\tilde{S})$ is nearly tight — it just has an additional $\lg\lg P$ 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 [32], 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. In fact, one can generate augmented DAGs for which it would be impossible to simultaneously provide good speedup and space bounds; the construction is similar to the lower bound in Section-3.1 in [16]. Since our replay scheduler provides an good speedup, it can not guarantee low space usage.

5. EMPIRICAL EVALUATION

This section empirically evaluates PORRidge. The main benefit of a processor-oblivious record-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 happens-before edges via section IDs in a schedule-independent fashion; during replay, PORRidge may need to suspend and resume strands upon lock acquires and releases.

	number	*number of lock acquires*				
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 acquires across all locks during execution. The min column shows the minimum number of lock acquires 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 acquires per lock and the standard deviation.

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 $3.73\times$ 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 [62] 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 [65] 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 Figure 1. Note that the characteristics during parallel execution may differ slightly for some of the graph benchmarks as they are non-

	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.

deterministic 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 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 acquire. 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 list to detect collisions of section IDs (discussed in Section 3), and the additional cache misses incurred by accessing the hash list incurred much larger overhead in these applications (8–9×). By reordering the bookkeeping data layout to obtain better spatial locality and using a bloom filter instead of a hash list, 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 acquires and releases. When a worker tries to acquire a lock, it must search the lock-acquire array or query the hash list with the current section ID to see if this lock acquire is the next in line to obtain the lock. If so, it can acquire the lock. Otherwise, it must suspend. Upon a release, the worker advances the head of the lock-acquire list; if the next lock acquire has been tried and suspended, the worker suspends its current execution and resumes the execution of the next lock acquire. 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-acquire arrays if there is ever any collisions in the section IDs. Thus, for most benchmarks, replay exhibits similar overhead as in recorded run when $P_{rec} = P_{rep} = 1$ as shown in Figure 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 acquires. 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

	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.

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 Figure 3. It turns out that for the most part, the additional overhead incurred by suspending and resuming lock acquires 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., $P_{rec} = 1$).

Recall the time bound for replay: its expected execution time on P workers is $O(W/P + \tilde{S} \lg \lg P)$, where W is the overall work in the computation and the $\tilde{S}$ is the span in the augmented dag. Since $S \leq \tilde{S} \leq S + B$, replay should scale as long as record scales if we ignore the $\lg \lg P$ 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 Figure **??**), indicating that it is generally safe to ignore the $\lg \lg P$ 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 explanation. 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 scales (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 list, replay must use either an array or a hash list. While arrays have fewer cache misses than the hash list, both of these have a larger memory footprint than the bloom filter. Indeed, recall that one of the optimizations that we implemented for replay is to use a lock-acquire array instead of a hash list if the number of lock acquires per lock is small (which is the case for these benchmarks). The overhead of replay was much higher when we used a hash list 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. How many 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.

5.5 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. For dynamic multithreaded computations, a processor-aware record-and-replay system would need to log additional information to record the inherent non-determinism in the scheduler, which would further increase the memory footprint of the recording.

Moreover, the strategy used by PORRidge has the additional benefit of scaling the replay beyond the number work-

				replay on P workers ($P_{rep} = P$) an execution recorded on P' workers ($P_{rec} = P'$)					
bench	*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.

ers used in record, which a processor-aware record and replay system cannot provide. If a recording is done on P workers and takes x time, in a processor-aware system, the replay cannot run in less than x time (asymptotically) no matter how many workers we give it. In fact, it would likely be slower since it would spin when it encounters a lock acquire that is not ready and would have to exactly replicate the steal patterns of the record, causing potentially more idleness. On the other hand, PORRidge never spins during replay and uses suspension to explore all the possible parallelism in the augmented dag. 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 $P_{rep} > P_{rec}$.

6. 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 reruns 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 [63] and JaRec [34] 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 happens-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 [43, 68]. 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 [2, 37, 48, 56]. Some systems track racy interleavings directly, which either add large overhead [42, 70], use coarse-granularity communication tracking (such as page-based conflict detection) that can be overly-conservative [29, 41], or rely on carefully modified virtual machines [20].

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 [57] is such a general

record and replay system based on Pin, a popular dynamic binary instrumentation framework [50], that captures all sources of nondeterminism including racy memory accesses, thread interleavings, and results from system calls. We ran PinPlay on Delaunay Refinement (`refine` described in Section 5) and find that it has $96.8\times$ overhead for recording and $16.1\times$ overhead for replay when executing the computation on one worker — 1-2 orders of magnitudes worse than the PORRidge overheads of $1.26\times$ and $1.16\times$, respectively. When we tried recording and replaying on multiple workers, the executions with PinPlay slowed down and showed no speedup. This result in part speaks to the performance advantage of PORRidge's approach, because PORRidge does not need to reproduce the runtimes non-determinism while traditional thread-based record-and-replay systems must. Reproducing the runtimes non-determinism requires logging all inter-thread interactions among worker threads and causing 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. 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 [36, 53, 58, 69], 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 [12, 15, 19, 22, 54, 61] 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, 54], most do not. Another approach is to enforce determinism through hardware [26, 27], compiler [10], OS [4, 11] or runtime approaches [49, 55]. 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 [30, 59] and in parallel [9, 51, 60, 67]. Some of these have led to implementations [30, 60, 67]. 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 [35, 64] or use sampling to determine where in the code causes workers to be idle [66].

Work-stealing Runtime with Multiple Deques. Prior work-stealing designs have used more than P deques for supporting concurrent data structures [1, 67] or blocking I/O operations [52, 71]; some provide theoretical scheduling bounds [1, 52, 67], but their modifications are for a different purpose and require different modifications and analyses.

7. CONCLUSION

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 happens-before relationships; for instance, `try-lock` and `compare-and-swap`. These require rethinking the exact semantics we want from a happens-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 happens-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.

Acknowledgments

This research was supported in part by National Science Foundation grants CCF-1150036, CCF-1218017, XPS-1439062,

CCF-1150013, CCF-1439126, CCF-1527692, and Department of Energy grant DE-SC0010295. We thank the referees for their excellent comments.

References

[1] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. 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, pages 84–95, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2821-0. doi: 10.1145/2612669.2612688. URL `http://doi.acm.org/10.1145/2612669.2612688`.

[2] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. URL `http://doi.acm.org/10.1145/1629575.1629594`.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2010.

[5] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 198–205, May 1999.

[6] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, Mar. 2009. URL `http://dx.doi.org/10.1109/TPDS.2008.105`.

[7] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal Comput.*, 29(1):180–200, Sept. 1999. ISSN 0097-5397. doi: 10.1137/S0097539795288490. URL `http://dx.doi.org/10.1137/S0097539795288490`.

[8] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. 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, pages 735–736, Orlando, Florida, USA, 2009. ACM. ISBN 978-1-60558-768-4.

[9] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 133–144, 2004.

[10] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, 2010. doi: http://doi.acm.org/10.1145/1736020.1736029.

[11] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 177–191, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924956`.

[12] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 81–96, 2009. doi: http://doi.acm.org/10.1145/1640089.1640096.

[13] C. Bienia and K. Li. Characteristics of workloads using the pipeline programming model. In *Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture*, June 2010.

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

[15] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[16] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 1995. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5): 720–748, Sept. 1999.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[19] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *USENIX Conference on Hot Topics in Parallelism*, pages 4–9, 2009.

[20] M. D. Bond, M. Kulkarni, M. Cao, M. Fathi Salmi, and J. Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *ACM International Conference on Principles and Practice of Programming in Java*, pages 90–101, 2015.

[21] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, USA, 3rd edition, 2015.

[22] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 691–707, Reno/Tahoe, Nevada, USA, Oct. 2010. URL `http://doi.acm.org/10.1145/1869459.1869515`.

[23] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. 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, pages 51–61, 2011.

[24] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005. URL `http://doi.acm.org/10.1145/1094811.1094852`.

[25] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, 1998.

[26] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2009. doi: http://doi.acm.org/10.1145/1508244.1508255.

[27] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–78, 2011. doi: http://doi.acm.org/10.1145/1950365.1950376.

[28] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, England, 1968. Originally published as Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[29] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 121–130, 2008. doi: http://doi.acm.org/10.1145/1346256.1346273.

[30] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA*, 1997.

[31] J. T. Fineman and C. E. Leiserson. Race detectors for Cilk and Cilk++ programs. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1706–1719. Springer, 2011.

[32] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[33] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, 2009.

[34] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A Portable Record/Replay Environment for Multi-threaded Java Applications. *Software Practice & Experience*, 34(6):523–547, 2004. ISSN 0038-0644. doi: 10.1002/spe.579.

[35] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, pages 145–156, Santorini, Greece, June 13–15 2010. doi: 10.1145/1810479.1810509.

[36] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *Communications of the ACM*, 52:93–100, 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1516046.1516068.

[37] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *ACM Conference on Programming Language Design and Implementation*, pages 141–152, 2013. doi: 10.1145/2491956.2462167.

[38] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[39] TBB. *Intel(R) Threading Building Blocks*. Intel Corporation, 2009. Available from `http://www.threadingbuildingblocks.org/documentation.php`.

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

[41] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 155–166, 2010. ISBN 978-1-4503-0038-4. doi: 10.1145/1811039.1811057. URL `http://doi.acm.org/10.1145/1811039.1811057`.

[42] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36:471–482, 1987. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1987.1676929.

[43] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–90, 2010. doi: http://doi.acm.org/10.1145/1736020.1736031.

[44] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *ACM Conference on Programming Language Design and Implementation*, pages 463–474, 2012. doi: 10.1145/2254064.2254119.

[45] I.-T. A. Lee and T. B. Schardl. 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)*, pages 111–122, Portland, OR, USA, June 2015. doi: 10.1145/2755573.2755599.

[46] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[47] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*, 2012.

[48] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via Tightly Bounded Recording. In *ACM Conference on Pro-*

gramming Language Design and Implementation, pages 55–64, 2015. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738001.

[49] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *ACM Symposium on Operating Systems Principles*, pages 327–336, 2011. doi: http://doi.acm.org/10.1145/2043556.2043587.

[50] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200, Chicago, IL, USA, June 11–15 2005. doi: 10.1145/1065010.1065034.

[51] J. Mellor-Crummey. 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)*, pages 24–33, Albuquerque, NM, USA, Nov. 18–22 1991. doi: 10.1145/125826.125861.

[52] S. K. Muller and U. A. Acar. 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, pages 71–82, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4210-0. doi: 10.1145/2935764.2935793. URL `http://doi.acm.org/10.1145/2935764.2935793`.

[53] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2006. doi: http://doi.acm.org/10.1145/1168857.1168886.

[54] D. Nguyen, A. Lenharth, and K. Pingali. 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, pages 499–512, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541964. URL `http://doi.acm.org/10.1145/2541940.2541964`.

[55] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, 2009. doi: http://doi.acm.org/10.1145/1508244.1508256.

[56] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *ACM Symposium on Operating Systems Principles*, pages 177–192, 2009. doi: http://doi.acm.org/10.1145/1629575.1629593.

[57] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. 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, pages 2–11, Toronto, Ontario, Canada, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772958. URL `http://doi.acm.org/10.1145/1772954.1772958`.

[58] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *IEEE/ACM International Symposium on Microarchitecture*, pages 576–585, 2009. doi: http://doi.acm.org/10.1145/1669112.1669183.

[59] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-16611-2.

[60] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. 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, pages 531–542, 2012.

[61] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20:483–545, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/291889.291893.

[62] A. D. Robison. 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.

[63] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17:133–152, 1999. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/312203.312214.

[64] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson. The Cilkprof scalability profiler. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*, SPAA '15, pages 89–100, Portland, Oregon, USA, June 2015. doi: 10.1145/2755573.2755603.

[65] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, 2012.

[66] N. R. Tallent and J. M. Mellor-Crummey. 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, pages 229–240, Raleigh, NC, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504210. URL `http://doi.acm.org/10.1145/1504176.1504210`.

[67] R. Utterback, K. Agrawal, J. T. Fineman, and I.-T. A. Lee. 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, pages 83–94, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4210-0. doi: 10.1145/2935764.2935801. URL `http://doi.acm.org/10.1145/2935764.2935801`.

[68] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ACM International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, pages 15–26, 2011. doi: http://doi.acm.org/10.1145/1950365.1950370.

[69] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *ACM/IEEE International Symposium on Computer Architecture*, pages 122–135, 2003. doi: http://doi.acm.org/10.1145/859618.859633.

[70] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object Centric Deterministic Replay for Java. In *USENIX Annual Technical Conference*, pages 30–30, 2011.

[71] C. S. Zakian, T. A. K. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton. *Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++*, pages 73–90. Springer International Publishing, Cham, 2016. ISBN 978-3-319-29778-1. URL `http://dx.doi.org/10.1007/978-3-319-29778-1_5`.

A. Artifact Evaluation

PORRidge is open source and currently available at `https://gitlab.com/wustl-pctg-pub/porridge.git`. 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 gitlab repository to help us continually improve the project.