Processor-Oblivious Record and Replay

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.

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, 37, 38, 42, 43, 45, 49, 53, 56, 57, 62, 67–69]. 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 replayed 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 [17, 33, 41], subsets of OpenMP [6], Threading Building Blocks [40], the Habanero family [8, 23], Task Parallel Library [47], 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 responsible 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.

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 constructs such as spawn and sync 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 pro-

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

[2] While data race freedom may seem to be a strong constraint, we note three things: (i) DRF is a common assumption for record and replay systems [34, 62], as well as other dynamic analyses [55]; (ii) there are a wide variety of robust, effective race detection tools for Cilk [25, 30, 31, 46]; and (iii) a racy program can be converted to a DRF program for the purposes of record and replay using techniques such as those used by Chimera [44].

gram can be replayed on more processors than the original recorded run!

The key insight behind PORRidge is as follows: there are multiple sources of non-determinism in scheduling when we execute a dynamic multithreaded program, for instance, the random work stealing decisions that the scheduler makes. However, for a data-race free computation, a recording run need not record all this information to reproduce it faithfully during replay; it is sufficient to just record the order in which various critical sections acquired a shared lock. To be more precise, a dynamic multithreaded program can be viewed as a directed acyclic graph, 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, if W is the ***work*** required by a parallel computation — its serial execution time — S is the ***span*** (or ***critical path length*** — longest sequence of dependencies in the computation, P is the number of processors, and B is the amount of work in critical sections, 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 [41] 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 (about 10%).

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 [18, 33] dynamically load balances a parallel computation across available ***worker*** threads. Each worker maintains a ***deque***, double-ended queue; 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 [18].

Modeling Critical Sections. Since our computations contain critical sections, we must model those. Each critical section of length x is simply a strand (chain) of x unit time nodes in the dag. Since it is a strand, there are no parallelism constructs within a critical section — therefore, 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 $B = \sum_i B_i$.

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. 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

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, which is stored with the lock object itself. Conceptually, a lock object in PORRidge contains a pointer to the underlying lock defined by the POSIX pthread specification [39] and an ordered list of successful lock acquires to this lock. When a worker successfully acquires a lock, it simply adds the its currently executing strand to the end of the list. If the lock is not available, the worker spins. At the end of the recorded execution, every lock object writes out the strands in the list to a log file in the order inserted.

Identifying Strands. For processor-oblivious replay, the information stored in the list must uniquely identify the strands in the computation dag and the identification must be consistent across executions. Here, we use ***pedigree*** [48], 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.

The open-source Cilk Plus runtime [41] readily provides support for pedigree; however, each read to the pedigree incurs a worst-case $\Theta(d)$ overhead, where d is the maximum ***spawn depth***, the number of spawn statements nested on the stack during serial execution. Since the pedigree must be read in every lock 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 [48] to give each strand a ***strand 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; a pedigree always hashes to the same random number provided that we use the same seed. Moreover, two random numbers generated via two different pedigree have a low probability of collision [48]. Using a similar strategy as DOTMIX, we obtain unique strand IDs with constant overhead per lock acquire.

Storing Strand IDs. The drawback of strand IDs is the (rare) possibility of collision. To detect collisions efficiently, the runtime employs a ***hash-list*** (instead of a list), which is a hash table whose values form a list implicitly by adding a next pointer to each entry in the hash table. To disambiguate collision, whenever a worker tries to add a strand id which is already in the table, it marks the first strand with the same strand ID as "has collision," reads the full pedigree of the current strand and stores it in the hash-list. Finally, during recording, the runtime also keeps track of the head and the tail of the list with a lock object — the tail so that it can manage the implicit list and the head so that it can write the result to the log at the end of the execution.

At the beginning of the replay, the runtime reads in the previously recorded log and recreates the hash-list and maintains the invariant that the head pointer to the list node 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 re-

suming 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 should suspend 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 earlier tried and suspended). This worker must then resume this suspended critical section.

Lock Acquire. When a worker encounters a lock acquire, it checks the head pointer 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 strand id and marks the corresponding hash-list node to indicate that the corresponding strand has been tried and suspended, and suspends the execution. The worker then work steals. Note that a worker cannot spin wait on a lock acquire that is not ready since this can lead to deadlocks.

Lock Release. The worker first advances the head pointer and checks to see if the next strand has been tried and suspended. If not, the worker simply continues the execution after the lock release. If the next strand 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 strand 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 Strand ID Collisions. A worker may encounter a head list node marked as "has collision," — in this case, multiple strands with the same id acquired this lock. Recall that during recording, the runtime stores full pedigrees only when it discovers a collision; therefore, the first strand involved in the collision has the strand id stored and the remaining strands involved in the collision have full pedigrees stored. Thus, if a worker finds a head node with the same strand ID but marked as "has collision," it must read its full pedigree and compare it against other list nodes hashed with the same strand IDs (for which the full pedigree is stored). If none of them match the current pedigree, it can 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 at 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 worker can suspend execution when it has 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, 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.

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 by the following mechanisms: (1) When a worker w suspends its currently active deque, it picks two workers uniformly at random and gives the deque to the worker with the smaller load; and (2) on every steal attempt, the thief looks at two workers, takes a suspended deque from the worker with a larger load and gives it to the worker with the smaller load. Second, 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.

In PORRidge implementation, we also do an optimization that is not necessary for the bound, but which improves performance in practice. 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 deques. When another worker steals from this deque, 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 $W/P + O(S + \lg 1/\varepsilon)$ with probability $1 - 1/\varepsilon$.*

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 $O(PS + P\lg 1/\varepsilon)$ with probability at least $(1 - 1/\varepsilon)$. 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 $O(W/P + \lg\lg P\tilde{S})$.*

As with the analysis of record, 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* $3^{\tilde{S}}$ *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* $O(P\lg\lg P\tilde{S})$.

The following lemma follows from Lemmas 6 and 15

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

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

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

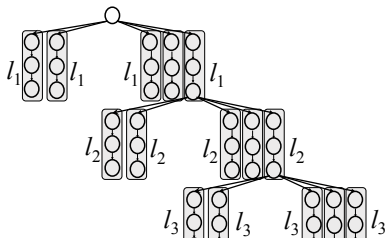

Figure 1: An example DAG with multiple locks where getting a tight bound for recording is impossible for an online scheduler. The offline scheduler can always schedule the "important" (in this case, the right-most) critical section first, but an online scheduler has no way of knowing which critical section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

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 [16], this overhead does not affect computations with sufficient parallelism.

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 strand IDs in a schedule-independent fashion; during replay, PORRidge may need to suspend and resume strands upon lock acquires and releases.

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, as long as the computation has enough parallelism, the record phase scales similarly as the baseline. When the baseline runs low on parallelism, however, the record phase can incur high overhead and cause slow down. The replay phase

	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.

tends to incur similar and sometimes smaller overhead than the recording. Due to its non-blocking execution model, replay using the same number of workers as in the recorded execution tends to execute faster than recording. In fact, as long as the recorded execution contains sufficient parallelism, the replay continues to get speedup beyond P workers, where P 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 [61] that solves a chess puzzle — given eight chess pieces excluding pawns, count the number of configurations where the pieces are placed such that they can attack all squares on an 8×8 chess board. The program uses reducers [32] to keep counts on the number of such configurations found and to perform I/O; we modified the program to use locks instead. Two benchmarks, `dedup` and `ferret`, from the PARSEC benchmark suite [13, 14] are used; they can be implemented as Cilk Plus programs that utilize reducers for performing file I/O, which we replace with locks. Finally, we converted three nondeterministic versions of graph algorithms from the Problem Based Benchmark Suite [64] 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 nondeterministic by nature. The first three benchmarks use few locks, but still have plenty of lock acquires; however, they do a significant amount of work outside of critical sections. The three graph benchmarks use a much larger number of locks, since there is one lock per vertex. 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. Hyperthreading is disabled. For recorded runs, running times are in seconds as a mean of five runs. For a given number of workers, the one with the median running time is chosen to be used for the replay

	baseline	*record*	*replay*
`chess`	87.03	87.12 (1.00×)	86.92 (1.00×)
`dedup`	48.70	48.91 (1.00×)	48.70 (1.00×)
`ferret`	9.01	9.05 (1.00×)	9.06 (1.00×)
`matching`	4.06	39.14 (9.64×)	39.75 (9.79×)
`MIS`	1.69	14.46 (8.56×)	14.17 (8.38×)
`refine`	12.10	21.80 (1.80×)	20.96 (1.73×)

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.

runs. For replay runs, running times are in seconds as a mean of ten runs. For the most part, the standard deviation was within 5% of the mean for both record and replay. There are a few exceptions: graph algorithms have higher standard deviation during replay. In particular, replaying a one worker recording on multiple workers has high standard deviation; for instance, up to 20% for `matching`. We will explain the reason for this phenomenon in Section 5.2.

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 1–9.79× with a geometric mean of 2.30. 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 high 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. Still, this would suggest that the recording overhead per lock acquire and release is about 8–10× than the usual lock acquire and release.unusually high.

To better understand this overhead, we measure the breakdown of record overhead for these benchmarks. For every lock acquire, PORRidge (1) obtains the strand ID for the strand, (2) inserts the strand ID into the hash-list, and (3) if something is already in the bucket, it checks for strand ID collision. Getting the strand ID only incurs 18% additional overhead for `MIS` and 4% additional overhead for `matching` (other benchmarks incur even less for getting strand IDs). The rest of the overhead comes from logging lock acquire entries into the hash-list.

It turns out that all of the lock acquisition overhead can be explained by cache misses. These graph algorithms have large working sets and display very little locality in accessing data. The additional bookkeeping required by recording puts even more pressure on the memory hierarchy, and since the original computation displays no locality, accesses to the hash-list for record also have no locality. Thus, even though recording only involves a few additional memory references per lock acquire to access the hash-list, it can incur quite a bit more overhead due to the additional cache misses. We find that during the executions of `MIS` and `matching`, recording incurs about five times more L3 cache misses than the baseline, which largely explains the high overhead.

5.2 Overhead of Replay

Replay, like record, incurs overhead upon lock acquires and releases. When a worker tries to acquire a lock, it must access that lock's hash-list and query the hash-list with the current strand ID to see if this strand is the next in line to access this lock. If so, it can acquire the lock. Otherwise, it must suspend. Upon a release, the worker advances the head of the hash-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.

If we record on one worker and replay on one worker, the execution 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 reads the hash-list instead of writing to it. We see from Figure 3 that in this case, as expected, replay exhibits essentially the same overhead as record.

The more interesting case is when we record on more than one worker and then replay on one worker ($P_{rec} > 1$ and $P_{rep} = 1$). In this case, replay has additional overheads — namely the overhead of suspending and resuming lock acquires. Figure 3 shows the overhead of replay on one worker, replaying executions recorded on different number of workers. We can compare these with the execution when we record one worker and replay on one worker (the first column of the figure). It turns out that for the most part, the additional overhead incurred by processor-oblivious replay is small. Besides `chess`, which executes faster on replay (a behavior we explain below), only a few data points incur slightly more than 10% additional overhead compared to one-worker record, such as `matching` and `MIS`. 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 this worker encounters a critical section that cannot be executed yet it must suspend its current deque and work steal. In addition, since work stealing is random, the next critical section it acquires may again not be the right one. Therefore, the worker may suspend many deques before encountering a critical section it can execute. Since these graph benchmarks have very little work outside critical sections and a very large number of lock acquires, these suspensions

	replay on one, recorded on P					
application	*P = 1*	*P = 2*	*P = 4*	*P = 8*	*P = 12*	*P = 16*
chess	86.92 (1.00×)	83.12 (0.95×)	68.32 (0.78×)	66.46 (0.76×)	69.29 (0.80×)	68.03 (0.78×)
dedup	48.70 (1.00×)	49.16 (1.01×)	48.91 (1.00×)	48.90 (1.00×)	48.99 (1.00×)	48.98 (1.00×)
ferret	9.06 (1.00×)	9.10 (1.01×)	9.10 (1.01×)	9.11 (1.01×)	9.08 (1.00×)	9.09 (1.00×)
matching	39.75 (1.02×)	40.67 (1.03×)	41.91 (1.07×)	43.28 (1.11×)	39.38 (1.01×)	44.72 (1.14×)
MIS	14.17 (0.98×)	14.52 (1.00×)	14.8 (1.02×)	14.74 (1.02×)	15.48 (1.07×)	16.02 (1.11×)
refine	20.96 (0.96×)	22.34 (1.03×)	22.18 (1.02×)	23.02 (1.06×)	22.72 (1.04×)	23.48 (1.08×)

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. Highlighted cells indicate execution time differ from that of the recorded run on one worker by more than $\pm 10\%$.

(and the cost of resuming later) cause these overheads.

This also explains why replay executions have high variance in running time. Since which deque to steal from is chosen at random, some times replay may get lucky (or unlucky) in choosing its steal attempt leading to smaller or larger number of suspensions. For this reason, we see a larger standard deviation in execution times — up to 20% for `matching` and between 10–15% for `MIS` when executions are recorded on $P \geq 2$ and replayed on one worker.

In principle, the same high execution variance should arise any time $P_{rep} < P_{rec}$. However, we find that when $P_{rep} > 1$, even though workers may not be able to find the critical sections that need to execute next, because multiple workers are expanding different parts of the augmented dag, there is less execution variance.

We now turn to the first row of Table 2, which shows that replaying for `chess` tends to be faster than the recorded run on one worker. This is strange, since replay does strictly more work than record. This is due to data access patterns in `chess`. It turns out that expanding the dag in the breadth-first fashion (i.e., the "help-first" execution schedule as opposed to "work-first" [35]) helps with the locality. If we record on multiple workers and replay on one worker, the replay execution is closer to the help-first schedule and therefore gets this benefit of locality. We measured the cache misses for the `chess` execution. Replay with $P_{rep} = 1$ incurs more than $2\times$ L3 cache misses than replay with $P_{rep} > 1$. This effect also shows up in the baseline and recording: if we execute `chess` baseline and record on multiple workers, it gets superlinear speedup, since multiple worker execution incurs fewer L3 cache misses than a single worker execution.

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 scalability profile for record tracks that of the baseline closely when the application has enough parallelism, as is the case for `chess`, `dedup`, and `ferret`. For applications that do not have as much parallelism, however, the record starts to incur high overhead and causes slow down, such as in the case of `matching`, `MIS`, and `refine` for $P_{rec} \geq 12$.

To explain this, we measured the cycles that workers spend waiting for a lock, and the cycles that workers spend in the critical sections. These values tend to increase substantially for record when going from eight workers to twelve; for baseline, these values also increase when we go from eight workers to twelve workers, but not as much as they do for record. This phenomenon can be traced back to the same locality issue we pointed out in Section 5.1. Since record incurs a larger number of cache misses compared to the baseline (due to the bookkeeping involved in recording), as we increase the number of cores, the pressure on the memory hierarchy increases, slowing down each critical section even further. In addition, when we execute on more than eight cores, more memory references are served off-socket further increasing execution time of critical sections.

This phenomenon affects the recording time in two ways: First, since the execution time of critical sections increases, the work W of the computation increases as we increase the number of processors. Therefore, according to the bound in Section 4, the recording time increases. Second, not only do critical sections execute for longer times, they also wait longer to get locks. That is, workers spend a lot of time spinning on a lock (this is captured in the B term in the bound), further increasing the execution time.

5.4 Scalability of Replay

Table 4 as well 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.

Recall the time bound for replay: its expected execution time on P workers is $O(W/P + \lg\lg P\tilde{S})$, where W is the overall work in the computation and the $\tilde{S}$ is the span in the augmented dag. Since $\tilde{S} \leq S + B$, replay should scale as long as record scales if we ignore the $\lg\lg P$ term. The experiments do indicate that it is generally safe to ignore the $\lg\lg P$ term and that the overheads of suspending and restarting in replay is small. For applications where the recording scales, indeed we see that the replay on the same number of workers scale similarly, as shown in the highlighted cells in Table 4 (`chess` is a counterexample which we explain below). Moreover, for applications where the recording does not scale, the replay continues to scale. In fact, the scalability of replay running on the same number of workers as in the recorded run gets better speedup than the recorded run, tracking closer to the scalability seen on the baseline.

Recall, as we explained in Section 5.3, that recording often does not scale (and sometimes experiences slowdown) on graph applications due to the fact that critical sections start taking longer due to increased pressure on memory hierarchy as the number of workers increases, causing an in-

bench	P	*baseline*	*record*	*replay on P workers ($P_{rep} = P$) an execution recorded on P' workers ($P_{rec} = P'$)*											
				$P' = 1$		$P' = 2$		$P' = 4$		$P' = 8$		$P' = 12$		$P' = 16$	
chess	1	87.03	(1.00×)	87.12	(1.00×)	86.92	(1.00×)	83.12	(1.00×)	68.32	(1.00×)	66.46	(1.00×)	69.29	(1.00×)	68.03	(1.00×)
	2	42.46	(2.05×)	42.49	(2.05×)	31.81	(2.73×)	31.96	(2.60×)	32.37	(2.11×)	31.52	(2.11×)	32.54	(2.13×)	32.43	(2.10×)
	4	19.19	(4.54×)	18.79	(4.64×)	16.41	(5.30×)	15.90	(5.23×)	15.78	(4.33×)	15.74	(4.22×)	15.77	(4.39×)	15.80	(4.31×)
	8	8.55	(10.18×)	8.59	(10.14×)	8.18	(10.63×)	8.00	(10.39×)	8.01	(8.53×)	7.97	(8.34×)	7.87	(8.80×)	7.87	(8.64×)
	12	5.48	(15.88×)	5.55	(15.70×)	7.87	(11.04×)	7.72	(10.77×)	7.46	(9.16×)	5.69	(11.68×)	5.75	(12.05×)	5.65	(12.04×)
	16	4.05	(21.49×)	4.14	(21.04×)	6.43	(13.52×)	6.12	(13.58×)	6.08	(11.24×)	4.39	(15.14×)	4.39	(15.78×)	4.31	(15.78×)
dedup	1	48.70	(1.00×)	48.91	(1.00×)	48.70	(1.00×)	49.16	(1.00×)	48.91	(1.00×)	48.90	(1.00×)	48.99	(1.00×)	48.98	(1.00×)
	2	25.99	(1.87×)	25.16	(1.94×)	25.42	(1.92×)	25.17	(1.88×)	25.11	(1.95×)	25.03	(1.95×)	25.05	(1.96×)	25.19	(1.94×)
	4	13.24	(3.68×)	13.07	(3.74×)	13.25	(3.68×)	13.35	(3.68×)	13.58	(3.60×)	13.10	(3.73×)	13.05	(3.75×)	13.06	(3.75×)
	8	6.61	(7.37×)	7.00	(6.99×)	7.81	(6.24×)	7.80	(6.30×)	7.64	(6.50×)	7.27	(6.73×)	7.42	(6.60×)	7.41	(6.61×)
	12	4.35	(11.20×)	4.94	(9.90×)	6.00	(8.12×)	5.91	(8.32×)	5.74	(8.52×)	5.44	(8.99×)	5.14	(8.53×)	5.32	(9.21×)
	16	3.39	(14.37×)	3.89	(12.57×)	5.08	(9.59×)	4.93	(9.97×)	4.81	(10.17×)	4.55	(10.75×)	4.39	(11.16×)	4.16	(11.77×)
ferret	1	9.10	(1.00×)	9.05	(1.00×)	9.06	(1.00×)	9.10	(1.00×)	9.10	(1.00×)	9.11	(1.00×)	9.08	(1.00×)	9.09	(1.00×)
	2	4.96	(1.83×)	4.92	(1.84×)	4.93	(1.84×)	4.96	(1.83×)	4.95	(1.84×)	4.96	(1.84×)	4.96	(1.83×)	4.98	(1.83×)
	4	2.55	(3.57×)	2.55	(3.55×)	2.56	(3.54×)	2.57	(3.54×)	2.57	(3.54×)	2.57	(3.54×)	2.57	(3.53×)	2.58	(3.52×)
	8	1.40	(6.50×)	1.42	(6.37×)	1.38	(6.57×)	1.40	(6.50×)	1.40	(6.50×)	1.41	(6.46×)	1.40	(6.49×)	1.39	(6.54×)
	12	1.03	(8.83×)	1.01	(8.96×)	1.02	(8.88×)	1.03	(8.83×)	1.03	(8.83×)	1.01	(9.02×)	1.03	(8.82×)	1.02	(8.91×)
	16	0.83	(10.96×)	0.81	(11.17×)	0.83	(10.92×)	0.82	(11.10×)	0.83	(10.96×)	0.83	(10.98×)	0.83	(10.94×)	0.82	(11.09×)
matching	1	4.06	(1.00×)	39.14	(1.00×)	39.75	(1.00×)	40.67	(1.00×)	41.91	(1.00×)	43.28	(1.00×)	39.38	(1.00×)	44.72	(1.00×)
	2	2.92	(1.39×)	34.40	(1.14×)	38.33	(1.04×)	35.01	(1.16×)	36.12	(1.16×)	33.21	(1.30×)	35.68	(1.10×)	36.75	(1.22×)
	4	1.75	(2.32×)	17.34	(2.26×)	15.74	(2.53×)	15.25	(2.67×)	15.87	(2.64×)	15.98	(2.71×)	14.88	(2.65×)	14.70	(3.04×)
	8	1.18	(3.44×)	10.16	(3.85×)	9.22	(4.31×)	8.89	(4.57×)	9.09	(4.61×)	8.68	(4.99×)	7.12	(5.53×)	7.66	(6.48×)
	12	0.55	(7.38×)	15.60	(2.51×)	9.73	(4.09×)	9.18	(4.43×)	8.48	(4.94×)	7.80	(5.55×)	7.33	(5.37×)	7.67	(5.83×)
	16	0.44	(9.23×)	15.90	(2.46×)	9.80	(4.06×)	9.39	(4.33×)	8.20	(5.11×)	7.31	(5.92×)	7.03	(5.60×)	6.90	(6.48×)
MIS	1	1.69	(1.00×)	14.46	(1.00×)	14.17	(1.00×)	14.52	(1.00×)	14.80	(1.00×)	14.74	(1.00×)	15.48	(1.00×)	16.02	(1.00×)
	2	1.48	(1.14×)	14.94	(0.97×)	12.26	(1.16×)	12.44	(1.17×)	11.53	(1.28×)	11.62	(1.27×)	11.15	(1.39×)	10.66	(1.50×)
	4	1.00	(1.69×)	7.63	(1.90×)	8.62	(1.64×)	7.12	(2.04×)	6.30	(2.35×)	6.00	(2.46×)	4.80	(3.23×)	4.96	(3.23×)
	8	0.80	(2.11×)	5.21	(2.78×)	7.62	(1.86×)	6.50	(2.40×)	4.76	(3.11×)	4.20	(3.51×)	3.15	(4.91×)	2.68	(5.98×)
	12	0.25	(6.76×)	8.92	(1.62×)	6.22	(2.28×)	5.70	(2.55×)	4.41	(3.36×)	3.19	(4.62×)	2.97	(5.21×)	2.66	(6.02×)
	16	0.24	(7.04×)	10.70	(1.35×)	7.83	(1.81×)	6.38	(2.28×)	4.69	(3.16×)	3.47	(4.25×)	3.01	(5.14×)	2.54	(6.31×)
refine	1	12.10	(1.00×)	21.80	(1.00×)	20.96	(1.00×)	22.34	(1.00×)	22.18	(1.00×)	23.02	(1.00×)	22.72	(1.00×)	22.48	(1.00×)
	2	8.05	(1.50×)	19.90	(1.10×)	16.74	(1.25×)	17.08	(1.31×)	17.40	(1.27×)	16.90	(1.36×)	17.08	(1.33×)	16.70	(1.41×)
	4	5.00	(2.42×)	14.60	(1.49×)	10.91	(1.92×)	10.35	(2.16×)	10.16	(2.18×)	10.31	(2.23×)	10.19	(2.23×)	10.58	(2.22×)
	8	3.54	(3.42×)	11.00	(1.98×)	7.81	(2.68×)	7.87	(2.84×)	7.74	(2.87×)	7.63	(3.02×)	6.78	(3.35×)	7.29	(3.22×)
	12	3.21	(3.77×)	19.80	(1.10×)	8.28	(2.53×)	7.82	(2.86×)	7.50	(2.96×)	7.52	(3.06×)	7.42	(3.06×)	7.47	(3.14×)
	16	2.90	(4.17×)	26.40	(0.83×)	8.48	(2.47×)	7.94	(2.81×)	7.93	(2.80×)	7.43	(3.10×)	7.35	(3.09×)	7.16	(3.28×)

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.

crease in both work W and spinning time. Replay has a similar limitation with respect to increase in the execution time of critical sections — therefore, the work increases for replay as well. However, recall that when a worker is blocked on a lock during recording, it simply spins and does not find other work to do; therefore longer critical sections increase waiting time. In the case of replay, workers never spin. When a lock is unavailable, workers suspend their deque and find other work to do. Therefore, replay runs are able to exploit all the parallelism that is present in the augmented dag and can scale even when record does not scale.

Note that `chess` seems to lack scalability on replay. This is an artifact of the phenomenon we explained in Section 5.2. Notice that record and replay on the same number of processors have the same execution time. So the lower scalability of replay is a denominator effect — since replay on 1 worker is much faster than record on 1 worker (due to breadth-first scheduling in replay getting better locality), it looks like replay scales less than record does. In fact, record is getting super-linear speedup when it runs in parallel, since parallel execution does more breadth-first execution.

Finally, note that, replay of an execution recorded on $P_{rec} = P$ workers can continue to scale when $P_{rep} > P$, as long as there is still parallelism in the augmented dag, as predicted by our theoretical analysis. Such scalability is evident across all benchmarks, as shown by the numbers in the same column below the highlighted cells in Table 4.

5.5 Benefits of Processor Obliviousness

While recording has high overheads and sometimes does not scale beyond a socket due to lack of locality and large memory footprint of some benchmarks, these overheads are not due to the processor-obliviousness of our strategy. The main difference between a processor aware recording and a a processor oblivious recording is what a worker records on each lock acquire. In a processor-aware recording, they would simply record the thread id; in PORRidge, it must get the pedigree of the strand and record it instead. However, as the data shows, getting the pedigree is not the cause of the overhead. The expensive part is simply logging information in the hash table — a processor aware recording would also have to write the thread id into some hash table like data structure. In addition, 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.

The scalability of replay is perhaps the biggest advantage

of the strategy used by PORRidge. A non-processor oblivious record and replay scheduler can not provide scalability beyond the number of workers used during recording. If a recording is done on P workers and takes x time, in a processor-aware system, the replay can not run in less than x time no matter how many workers we give it. In fact, it would likely be slower since it would spin when unable to acquire locks 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 faster 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 re-runs the same threads with the same behavior. Recording and replaying on the same number of threads simplifies both the recording process (as thread-based identifiers can be used to identify operations) and the replay process (as there is no need to map operations from the recorded run onto a different number of threads). RecPlay [62] 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 [45, 67]. 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, 38, 49, 56]. Some systems track racy interleavings directly, which either add large overhead [43, 69], use coarse-granularity communication tracking (such as page-based conflict detection) that can be overly-conservative [29, 42], or rely on carefully modified virtual machines [20]. Chimera [44] uses static race detection to identify potentially-racing pairs of accesses, and uses lightweight synchronization, as well as lock coarsening, to enable a simple record and replay technique. Such an approach, could be adapted to make PORRidge applicable to racy Cilk programs.

Another strategy is to record information at the *hardware* level [37, 53, 57, 68], 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, 60] 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 [50, 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, 58] and in parallel [9, 51, 59, 66]. Some of these have led to implementations [30, 59, 66]. The parallel tools are generally processor-oblivious; a single run on any number of workers gives the correct answer. Another important class of tools is *performance profilers*, that either measure work and span of the program directly during execution [36, 63] or use sampling to determine where in the code causes workers to be idle [65].

Work-stealing Runtime with Multiple Deques. Prior work-stealing designs have used more than P deques for supporting concurrent data structures [1, 66] or blocking I/O operations [52, 70]; some provide theoretical scheduling bounds [1, 52, 66], 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. Finally, we can explore other mechanisms to enable processor-oblivious record and replay to see if some of them will give better performance.

8. 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.

[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.

[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.

[7] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal Comput.*, 29(1):180–200, Sept. 1999.

[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.

[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.

[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.

[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.

[13] 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.

[14] 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.

[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, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[17] 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.

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

[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.

[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.

[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.

[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.

[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.

[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, 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.

[33] 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.

[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.

[35] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Rome, Italy, 2009. IEEE Computer Society.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

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

[42] 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.

[43] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36:471–482, 1987.

[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.

[45] 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.

[46] 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.

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

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

[49] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via Tightly Bounded Recording. In *ACM Conference on Programming Language Design and Implementation*, pages 55–64, 2015.

[50] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.

[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.

[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.

[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.

[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.

[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.

[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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17:133–152, 1999.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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 Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, 2011.

[68] 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.

[69] 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.

[70] 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.