Fast and Fair Randomized Wait-Free Locks

Naama Ben-David
VMware Research
USA
bendavidn@vmware.com

Guy E. Blelloch
Carnegie Mellon University
USA
guyb@cs.cmu.edu

ABSTRACT

We present a randomized approach for wait-free locks with strong bounds on time and fairness in a context in which any process can be arbitrarily delayed. Our approach supports a tryLock operation that is given a set of locks, and code to run when all the locks are acquired. A tryLock operation, or attempt, may fail if there is contention on the locks, in which case the code is not run. Given an upper bound κ known to the algorithm on the point contention of any lock, and an upper bound L on the number of locks in a tryLock's set, a tryLock will succeed in acquiring its locks and running the code with probability at least $1/(\kappa L)$. It is thus fair. Furthermore, if the maximum step complexity for the code in any lock is T, the attempt will take $O(\kappa^2 L^2 T)$ steps, regardless of whether it succeeds or fails. The attempts are independent, thus if the tryLock is repeatedly retried on failure, it will succeed in $O(\kappa^3 L^3 T)$ expected steps, and with high probability in not much more.

CCS CONCEPTS

• **Theory of computation → Concurrent algorithms**.

KEYWORDS

locks, wait-freedom, randomized algorithm

ACM Reference Format:
Naama Ben-David and Guy E. Blelloch. 2022. Fast and Fair Randomized Wait-Free Locks. In *Proceedings of the 2022 ACM Symp. on Principles of Distributed Computing (PODC '22), July 25–29, 2022, Salerno, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3519270.3538448

1 INTRODUCTION

In concurrent programs, *locks* allow executing a 'critical section' of code atomically, so that it appears to happen in isolation. Locks are likely the most important primitives in concurrent and distributed computing; they give the illusion of a sequential setting, thereby simplifying program design. However, locks can also become scalability bottlenecks for concurrent systems.

To illustrate these concepts, we use the classic dining philosophers problem as a running example. In the *dining philosophers problem*, first introduced by Dijkstra, n philosophers sit around a table, with one chopstick between each pair of philosophers. Each philosopher is in one of three states – thinking, hungry or eating – when hungry, they need to pick up both adjacent chopsticks to be able to eat. When done eating, they put down the chopstick and think for an unpredictable amount of time before next being hungry. In the asynchronous setting, a scheduler decides when each philosopher takes a step, and can delay a philosopher arbitrarily. What should philosophers do to minimize the number of steps they take from becoming hungry until they are done eating? To avoid having the philosophers starve, we make two assumptions: firstly, once a philosopher acquires the chopsticks, the number of steps taken eating is bounded by a constant, and secondly, others can help a philosopher eat by taking steps on their behalf. Without the first, a philosopher could starve their neighbor by eating forever, and without the second, the scheduler could starve a philosopher by never letting its chopstick-holding neighbor take a step.

In this paper we present the first algorithm for this setting that ensures that each philosopher will acquire their chopsticks and eat in $O(1)$ steps in expectation. Clearly, our algorithm thus ensures progress is made quickly (at least in the asymptotic sense), and ensures fairness in the sense that everyone who is hungry gets to eat. Our algorithm is randomized, and assumes an oblivious adversarial scheduler (i.e., one that decides the interleaving of the philosopher's steps ahead of time), but adaptive adversarial philosophers who can choose how long to think knowing everything about the system.

We are not just interested in the dining habits of philosophers, but more generally in fine-grained locks. The chopsticks represent the locks, the philosophers processes, and eating represents a critical section of code. By allowing arbitrary code in the critical section, more than two locks per critical section, and arbitrary conflicts among the locks (not just neighbors on a cycle), the setting covers a wide variety of applications of fine-grained locks. For example, it captures operations on linked lists, trees, or graphs that require taking a lock on a node and its neighbors for the purpose of making a local update. Indeed, such local updates with fine-grained locks are the basis of a large number of concurrent data structures [10, 15, 21, 28, 32, 37, 40, 41], and of graph processing systems such as GraphLab [38]. Note that in many of these applications, the number of locks that need to be taken is still a small constant.

Our approach relies on light-weight *tryLock* attempts, which may fail and then be retried. Specifically, a tryLock specifies a set of locks to acquire, and code to run in the critical section. If a tryLock attempt by a process successfully acquires the locks, the given code is run. In this paper, critical sections can contain arbitrary code involving private steps along with reads, writes and CAS operations on shared memory.[1] We do not allow nesting of locks—i.e., the critical code cannot contain another tryLock. The critical section ends with a call to *release*, which releases all the process's locks. For mutual exclusion, we assume that if two processes have

PODC '22, July 25–29, 2022, Salerno, Italy

ACM ISBN 978-1-4503-9262-4/22/07…$15.00
https://doi.org/10.1145/3519270.3538448

[1] While most locks do not allow critical sections to experience races, we allow such scenarios, for more general group-locking mechanisms.

acquired the same lock, it must appear (based on updates to shared memory) that their critical sections did not overlap in time.[2]

Our main contribution is the following result.

Theorem 1.1 (Informal). *Let κ be the maximum number of tryLock attempts on any lock at any given time, let L be the maximum number of locks per tryLock attempt, and let T be the maximum number of steps taken by a critical section. There exists an algorithm for wait-free fine-granularity locks against an oblivious scheduler with the following properties.*

- ***Step Bound.*** *Each tryLock attempt takes $O(\kappa^2 L^2 T)$ steps.*
- ***Fairness Bound.*** *Each tryLock attempt of a process p succeeds with probability at least $\frac{1}{\kappa L}$ independently of p's other attempts, and allowing for a adaptive adversary to decide when to attempt the tryLock.*

Since attempts by a process p are independent, a direct corollary of this result is a wait-free fine-grained lock algorithm that succeeds in expected $O(\kappa^3 L^3 T)$ steps; simply retry upon failure. This is the first result that achieves any step complexity bounds that rely only on these parameters. As a special case, our results imply an $O(1)$ step solution to the dining philosophers problem; that is, assuming it takes constant steps to eat, each attempt to eat succeeds with probability 1/4 and takes $O(1)$ steps to complete (here, $\kappa = L = 2$).

We first describe an algorithm that assumes knowledge of the bounds κ and L, and then remove this assumption using a guess-and-double technique at the cost of a logarithmic loss in success probability. That is, for a tryLock with bounded contention κ that is *unknown* to the algorithm, the probability of success is $\Omega\left(\frac{1}{\kappa L \log(\kappa L T)}\right)$.

Achieving non-blocking progress requires processes to help each other complete their critical sections. This may result in several processes running the same code when helping concurrently. Thus, to ensure correctness, critical sections must be made *idempotent*, that is, regardless of how many times they are run, they appear to have only executed once. The notion of idempotence in computer science has been recognized as useful in various contexts [16, 19, 20]. Turek, Shasha and Prakash [48] and Barnes [9], designed lock-free locks and showed how to make any code based on non-concurrent reads and writes idempotent with constant overhead. However, they did not distill this property, and instead present their protocols as ad-hoc ways to use lock-free locks.

In this paper, we formally define idempotence for concurrent programs, and present a new and more general construction to achieve it. In particular, the Turek, Shasha and Prakash's, and Barnes' approaches only supported reads and writes in critical sections, and only if they did not race. We allow for races and also support CAS. As with theirs, our approach only has a constant factor overhead. Thus, our wait-free locks are applicable to general code without any asymptotic overhead. We believe that the definition of idempotence and its new construction are of independent interest; indeed our formalization of idempotence has recently been used elsewhere [13].

2 ASSUMPTIONS AND APPROACH

For the purpose of outlining the key assumptions, challenges and approach, and to generalize beyond dining philosophers while still abstracting away details of the machine model, we consider acquiring locks as a game involving players and competitions. Each player (tryLock attempt or philosopher) p participates in the game by specifying a set of competitions (locks or chopsticks) it wants to participate in. Different players can specify different (but potentially overlapping) sets of competitions. While a player is in the game, it takes steps to try to win its competitions, and a scheduler interleaves the steps of the different players. If a player wins all its competitions (acquires all its locks), it wins the game and celebrates (executes its critical section or eats). The celebration is itself a sequence of steps. It then exits. To ensure mutual exclusion, no two players can simultaneously have won their game (before exiting) if they share a competition.

Adversarial assumptions. An *adversarial scheduler* is often used to model the inherent asynchrony in concurrent systems; that is, the order in which processes execute steps (the *schedule*) is assumed to be controlled by an adversary. An *adaptive adversary* is assumed to see everything that has happened in the execution thus far (the history), whereas an *oblivious adversary* is assumed to make all of its scheduling decisions before the execution begins. Some separations are known between adaptive and oblivious scheduler settings [1, 24, 25, 42]. Often, an oblivious adversary is considered to be a reasonable assumption, since asynchrony in real hardware is not generally affected by the values written on memory.

In our setting, in addition to the adversarial scheduler, the players can be adversarial, possibly trying to increase or decrease the probability of celebrating. In the setting of tryLocks it is unreasonable to assume that the players are oblivious having pre-decided when to enter the game and with what competitions. This is because a player will likely need to try again if it fails on the first attempt, and possibly on a different set of locks. Therefore for all but the very simplest protocols the point at which a player requests to enter a game, and possibly which competitions it requests to compete on, will depend on what has happened so far. We therefore assume the player is adaptive and makes decisions knowning the full history. In summary, we assume two separate adversaries; the *player adversary*, which is *adaptive* and controls the start time and set of locks of each tryLock attempt, and the *scheduler adversary*, which is *oblivious* and controls the order of player steps.

Random Priorities. In our algorithm, as in other algorithms for similar problems [22, 36, 39, 42], each player is assigned a *random priority* such that higher priorities win over lower priorities. In the synchronous setting this by itself "solves" our problem. In particular in a round a set of players play the game by (1) picking their priority, (2) checking if they are the highest priority on all their competitions, and (3) celebrating if they are. Assuming independent and unique priorities, the probability of a player p celebrating is at least inversely proportional to the number of distinct players that requested any of the competitions p played on. The game is therefore fair, and assuming a synchronous scheduler (round-robin), the number of steps is bounded.

Unfortunately in the asynchronous setting, even with an oblivious scheduler, the situation is much more difficult. Firstly, bounding the steps requires having players help others; otherwise, a player could be blocked waiting for another player to celebrate. We solve this using idempotent code. Secondly, and much more subtilely,

[2] We say "appear" since helping can cause instructions to overlap in time, but those instructions will have no effect on the shared state.

keeping the competition fair is challenging against an adaptive player adversary. Most obviously, if the player adversary wants a player to loose it could wait for other strong players to be in shared competitions (recall that it can see the history), and then start the player. Even if we hide the priorities from the adversary, it could likely gain knowledge by how the players are doing. Even an oblivious player adversary could skew the game; if strong players take more steps than weak players and stay active longer, incoming players would see a biased field of strong players. There are several other subtle difficulties with achieving fairness.

Our approach. Our approach to making the game fair is to ensure the adversary's choices introduce no bias once a player enters the game. We prevent the introduction of biased priorities by the adversaries with two key ideas. Firstly, each player enters the game in a *pending* state, before its priority is assigned. Before being assigned a priority, a player p must help complete the attempts of all the competitors that started before it. That is, any player p' whose priority was known to the player adversary before player p joined the game will be forced to finish competing without competing against p. After completing this helping phase, p is assigned its priority, in what we call its *reveal* step; after this step, p is no longer pending, and is now *active*.

To implement the reveal step atomically, we model each competition (lock) as an *active set* object, which keeps track of membership (i.e. who is currently competing on this competition/lock). It allows players to insert and remove themselves, as well as query the object to get the set of currently active members. We then model the system of competitions as a *multi active set* object, which allows players to insert themselves into the membership of several competitions at once, i.e., all the competitions they compete on. The active state is then easy to implement; players are considered active if they appear in the active set of their locks.

However, helping others before becoming active is not sufficient to mitigate the bias that the adversary could introduce; the player adversary can enter a new player p' after p but while p is still pending, and the player p' could overtake p and become active before p (recall the scheduler adversary can make p' much faster than p). Based on steps in the protocol, p''s priority could affect the step at which p reaches its reveal step. This allows the adversary to affect whether p competes with p' based on the latter's priority.

To avoid this problem, our second idea is to force the player to stay in the pending state for a fixed number of its own steps, before revealing itself and competing. We do so by introducing *delays* in which the player simply stalls until it has taken enough steps since it joined the game. The important aspect of introducing these fixed delays is that the time a player becomes active (and thus begins to compete) is unaffected by other players. It therefore cannot be sucked earlier or pushed later based on the priority of current competitors.

We note that our approach is robust against strong adversarial players, but only an oblivious scheduler. A strong scheduler could still move the point at which a player reveals itself based on known priorities. We leave handling an adaptive scheduler adversary as an open question.

3 RELATED WORK

Randomization in Locks. Using randomization to acquire locks is a difficult problem that has been studied for many years. The difficulty arises from the lack of synchronization among processes, and the ability of the adversary to delay processes based on observations of the current competition. Rabin [42] first considered the problem for a single lock, and only for the acquisition (i.e. no helping). Like ours, his scheme used priorities. Saias [43] showed the algorithm did not satisfy the claimed fairness bounds due to information leaks of the sort described in Section 2, and Kushilevitz and Rabin [34] fixed it with a more involved algorithm. Lehmann and Rabin also developed an algorithm for the dining philosophers problem [36]. Lynch, Saias and Segala [39] later proved that with probability 1/16 within 9 rounds one philosopher would eat. However, our goal is much stronger, requiring a constant fraction to eat. Moreover, their model is not fully asynchronous—a round involves every process taking a step. Duflot, Fribourg, and Picaronny generalized the algorithm to the fully asynchronous setting [22], but at the cost of a bound that depends on the number of processors.

More recent work has also looked at randomized mutual exclusion for a single lock and without helping [24, 25]. This work has focused on the Distributed Shared Memory (DSM) model, which separates local from remote memory accesses. It shows that although the local time is necessarily unbounded (since there is no helping), the number of remote accesses can be bounded. Most of the above work has assumed an adversary that knows what instructions have been run on each process, but not the arguments of those instructions. This is more powerful than an oblivious adversary, but less powerful than an adaptive one. None of the work considered separating the player adversary from the scheduler adversary.

It has been shown that a tenet of concurrent algorithm design, linearizability [30] does not nicely extend when randomization is introduced [26]. Linearizability allows operations that take multiple steps to be treated as if they run atomically in one step. Unfortunately, however, analyzing probability distributions for the single step case does not generalize to a multistep linearized implementation, especially when analyzed for a weaker adversary. This is what lead to some of the difficulties encountered by the previous work, and some of the challenges we face.

The Need for Randomization. It seems unlikely that acquiring wait-free fine-granularity locks can be done in $O(1)$ steps deterministically. This is even true in the simpler case of the dining philosophers, where each philosopher only ever tries to acquire two locks and each lock only ever has two philosophers contending on it. Assuming the philosophers do not know their position around the table, this is even true in a *synchronous* setting; the best known solution to the similar two-ruling-set problem, i.e., identifying a subset of n philosophers who are separated by one or two other philosophers, takes $O(\log^* n)$ steps [18].

The issue is that there is a symmetry that needs to be broken. In the randomized synchronous setting, the problem becomes easy using random priorities as discussed in Section 2—every philopher will have the highest priority among itself and its two neighbors with probability 1/3 and will therefore eat with that probability. In the asynchronous setting, the problem can be solved in $O(n)$ steps deterministically using, for example, Herlihy's universal wait-free

construction [29]. Every philosopher can announce when they are hungry and then try to help all others in a round robin manner, using a shared pointer to the philosopher currently being helped. Using more sophisticated constructions [3], the steps can be reduced, but the total number of steps still depends on the total number of concurrently hungry philosophers in the system.

Lock-Free Locks. Turek et al. [48] and independently Barnes [9] introduced the idea of lock-free locks. They are both based on the idea of leaving a pointer to code to execute inside the locks, such that others can help complete it. In the locked code, Turek et al.'s method supports reads and writes and locks nested inside each other. As with standard locks, cycles in the inclusion graph must be avoided to prevent deadlocks. Their approach thus allows arbitrary static transactions via two-phase locking by ordering the locks, and acquiring them in that order. It uses recursive (or "altruistic") helping in that it recursively helps transactions encountered on a required lock. It is lock free, uses CAS, and although the authors do not give time bounds it appears that if all transactions take at most T_m time in isolation, the amortized time per transaction is $O(pT_m)$, where p is the number of processes. Barnes's approach supports arbitrary dynamic transactions in a lock free manner, and uses LL/SC. It uses a form of optimistic concurrency [33] allowing for dynamic transactions. As with the Turek et al. approach, it uses recursive helping. Neither approach is wait-free—a transaction can continuously help and then lose to yet another transaction.

To allow our locks to be non-blocking, we present a general construction for achieving idempotence. A similar construction was recently presented and used it to implement lock-free locks [13]. Their lock-free locks, while efficient in practice, do not have a bound on steps per tryLock attempt, as a single attempt can help arbitrarily many other ongoing attempts (not necessarily only on the locks in its lock set). They are lock-free, but not wait-free.

Contention Management in Transactions. Shavit and Touitou [46] introduce the idea of "selfish" helping in the context of transactions. They argue that if a transaction encounters a lock that is taken, it should help the occupant release this lock, but not recursively help. In particular, if while helping another transaction, it encounters a taken lock, then it aborts the transaction being helped. Their approach only supports static transactions, as it needs to take locks in a fixed order. It differs from our helping scheme in that there are no priorities involved. In our scheme, when a transaction being helped meets another transaction on a lock, we abort the one with lower priority and continue with the one being helped if it is not the one aborted. Shavit and Touitou's approach is again lock-free but not wait-free. The worst case time bounds are weaker than Turek et al. or Barnes since there can be a chain of aborted transactions as long as the size of memory, where only the last one succeeds.

Fraser and Harris [23] extend Barnes's approach based on optimistic concurrency and recursive helping. The primary difference is that they avoid locks for read-only locations by using a validate phase (as originally suggested by Kung and Robinson [33]). They break cycles between a validating read and a write lock on the same location, by giving arbitrary (not random) priorities to the transactions to break this cycle. As with Shavit and Touitou's method, operations can take amortized time proportional to the size of memory. There has been a variety of work on contention management for transactions under controlled schedulers, some of it using randomization [8, 27, 44, 45], but it does not apply to the asynchronous setting we are considering.

Efficient Wait-Freedom. Starting with Herlihy [29], many researchers have studied wait-free universal constructions, many of which can be applied to at least a single lock, but most of these have an $O(P)$ factor in their time complexity, where P is the *total number of processes in the system*, meaning that even under low contention they are very costly. Afek et al. [2] describe an elegant solution for a universal construction, or single lock in our terminology, that reduced the time complexity to be proportional to the point contention instead of the number of processors. Attiya and Dagan [7] describe a technique that should be able to support nested locks, although described in terms of operations on multiple locations. They only support accessing two locations (i.e., two locks). Considering the conflict graph among live transactions, they describe an algorithm such that when transactions are separated by at least $O(\log^* n)$ in the graph, they cannot affect each other. The approach is lock free, but not wait free, and no time bounds are given. Afek et al. [3] generalize the approach to a constant k locks (locations) and describe a wait-free variant using a Universal construction. They show that the step complexity (only counting memory operations) is bounded by a function of the contention within a neighborhood of radius $O(\log^* n)$ in the conflict graph. Both approaches are very complicated due to their use of a derandomization technique for breaking symmetries [18].

4 MODEL AND PRELIMINARIES

In this paper we use standard operations on memory including `Read`, `Write`, and `CAS`. Beyond memory operations, processes do local operations (e.g. register operations, jumps, ...). Whenever we discuss the execution *time* for a process, we mean **all** operations (i.e., instructions) that run on the process including the local ones.

A *procedure* is a sequencial procedure with an invocation point (possibly with arguments), and a response (possibly with return values). A *step* is either a memory operation or an invocation or response of a procedure. We assume all steps are annotated with their arguments and return values, and we say two steps are *equivalent* if these are the same. We say a memory operation has *no effect* if it does not change the memory (e.g. a read, a failed CAS or a write of the same value). We assume the standard definition of linearizability [30].

The *history* of a procedure is the sequence of steps it took, or has taken so far. The history of a concurrent program is some interleaving of the histories of the individual procedures. A history is valid if it is consistent with the semantics of the memory operations.

A *thunk* is a procedure with no arguments [31]. Note that any code (e.g., the critical section of a lock) can be converted to a thunk by wrapping it along with its free variables into a closure [47] (e.g., using a lambda in most modern programming languages). Here, for simplicity, we also assume thunks do not return any values—they can instead write a result into a specified location in memory. A thunk runs with some local private memory, and accesses the main memory via a fixed set of memory operations.

A *lock* object ℓ provides a $tryLock_\ell$ operation, which returns a boolean value; if false, we say the $tryLock_\ell$ fails, and if it returns

true, then we say that the $tryLock_\ell$ succeeded. We also define a general *tryLock* procedure whose arguments are a *lock set*, i.e., a set of lock and a *thunk*. We call the execution of a trylock a *tryLock attempt*. A tryLock calls $tryLock_\ell$ on each of the locks in its set, and succeeds if and only if all of them succeed. A tryLock attempt p returns a boolean value indicating whether it succeeded or failed, and satisfies mutual exclusion as defined in Definition 4.3.

In this paper, we aim to construct *randomized wait-free* locks, and furthermore bound running time and success probabilities in terms of the *point contention* on the locks in the system. We say an algorithm is *randomized wait-free* if each process takes a finite expected number of steps until its operation succeeds (see Chor et al [17] for a more formal definition). In this paper, the operations of processes are tryLock attempts. We say a tryLock attempt is *live* on a lock ℓ from its invocation to its response (inclusive) if ℓ is in its lock set. The *point contention of lock ℓ at time t* is the number of live tryLock attempts at time t that contain ℓ in their lock set. The *maximum point contention*, k_ℓ of lock ℓ is the maximum point contention ℓ can have at any point in time across all possible executions. We let κ be an upper bound on k_ℓ for all locks ℓ in the system. The contention of a tryLock attempt p, C_p, is the sum across all of p's locks of k_ℓ. That is, $C_p = \sum_{\ell \in p.lockList} k_\ell$.

We assume two adversaries; an *adaptive player adversary* and an *oblivious scheduler adversary*. Formally, the scheduler adversary is a function from a time step to the process that runs an instruction on that time step, which produces a *schedule*. Our algorithms do not know this function, and the scheduler can delay any given process for an arbitrary length of time. The player adversary is a function from the history of an execution and a given process to a boolean indicating whether the given process starts a new tryLock at its next step, and if so with which locks.

4.1 Idempotence

To allow processes to help each other complete their thunks (critical sections) on a lock, we must ensure that regardless of how many processes execute a thunk, it only appears to execute once. For this, we use the notion of *idempotence*, which roughly means that a piece of code that is applied multiple times appears as if was run once [12, 16, 19, 20]. We define the notion of idempotence here, and show how to make any thunk involving `Read`, `Write` and compare-and-swap (`CAS`) instructions into one that is idempotent with constant overhead in the full version of the paper [11]. Barnes [9] and Turek et al. [48] do not extract the notion of idempotence, but do describe a way to make code based on non-concurrent reads and writes idempotent under our definition (below).

A *run* of a thunk T is the sequence of steps taken by *a single process* to execute or help execute T. The runs for a thunk can be interleaved. A run is *finished* if it reached the end of T. We say a sequence of steps S is *consistent* with a run r of T if, ignoring process ids, S contains the exact same steps as r. We use $E \mid T$ to denote the result of starting from an execution E and removing any step that does not belong to a run of the thunk T.

Definition 4.1 (Idempotence). A thunk T is idempotent if in any valid execution E consisting of runs of T interleaved with arbitrary other steps on shared memory, there exists a subsequence E' of $E|T$ such that:

(1) if there is a finished run of T, then the last step of the first such finished run must be the end of E',
(2) removing all of T's steps from E other than those in E' leaves a valid history consistent with a single run of T.

The definition essentially states that the combination of all runs of a thunk T is equivalent to having run T once, and finishing at the response of the first run.

In the full version of the paper [11] we describe a simulation/-translation that converts thunks involving `Read`, `Write` and `CAS` instructions into one that is idempotent, proving the following result.

Theorem 4.2. *Any thunk using only* `Read`, `Write` *and* `CAS` *operations on shared memory can be simulated using* `Read`*s,* `Write`*s and* `CAS`*es as primitive operations such that (1) it is idempotent, (2) every simulated memory operation takes constant time, (3) the simulated operations are linearizable.*

We note that recent concurrent work proved a similar result [13], but our simulation technique is different and we believe it is of independent interest.

4.2 Mutual Exclusion with Idempotence

We say the *interval* of an idempotent thunk is from the first step of any run of the thunk until the last step of the first run that completes. When implementing a tryLock, the safety property we require is as follows.

Definition 4.3 (Mutual Exclusion with Idempotence). If a tryLock attempt $\mathcal{A}$ with thunk $\mathcal{T}$ and lock set $\mathcal{L}$ succeeds, then there is a run of $\mathcal{T}$ that executed to completion. Furthermore, $\mathcal{T}$'s interval does not overlap the interval of any other thunk whose lock set overlaps $\mathcal{L}$. If $\mathcal{A}$ fails, there is no run of $\mathcal{T}$.

5 A MULTI ACTIVE SET ALGORITHM

We now define the *multi active set* problem and present an algorithm that solves it using an *active set* object. Multi active sets will be useful in implementing our locking scheme; in Section 6, we show how to implement fast and fair locks by representing them as a multi active set object.

The *active set* object was first introduced by Afek et al. [4]. It has three operations; *insert*, *remove*, and *getSet*. A getSet operation returns the set of elements that have been inserted but not yet removed. Insert and remove operations simply return 'ack', and processes must alternate insert and remove calls.

The *multi active set* problem is a generalization of the active set problem to multiple sets. Instead of an insert, the data structure supports a *multiInsert* that inserts an *item* into a collection of sets. The *multiRemove* operation is with respect to the previous multiInsert operation, and removes the item from the sets it was inserted into. The getSet operation takes a set as an argument, and behaves the same as for the active set, returning all items in the given set.

5.1 Active Set Algorithm

We now present a novel linearizable active set algorithm. Its pseudocode appears in Algorithm 1. In the full version of the paper [11], we prove it correct and show that its step complexity is *adaptive* to

the size of the set; insert and remove operations take $O(k)$ steps for a set with k elements, and the getSet operation takes constant time.

An `announcements` array of C slots is maintained, where C is the maximum number of elements that can be in the set at any given time. Each slot has an `owner` element and a `set`, which is a pointer to a linked-list of elements. To insert an element, a process traverses the announcements array from the beginning, looking for a slot whose `owner` field is empty. It then takes ownership of this slot by CASing in its new element into the slot's owner field. To remove an element from slot i, a process simply changes the owner of slot i to `null`. We assume that a process maintains the index of the slot that it successfully owned in its last insert, and uses this index in its next call to remove. Intuitively, the owner fields of all the slots make up the current active set. An insert operation can always find a slot without an owner, since there are C slots.

To help implement an efficient linearizable getSet function, the insert and remove operations propagate the changed ownership of their slot to the top of the announcements array by calling the `climb` helper function. The `climb` function works as follows. Starting at the slot given as an argument, it traverses the announcements array to the top, replacing the `set` field of the current slot i with the `set` of the previous slot $i+1$, plus the owner of slot i. That is, the `climb` function intuitively collects all owners of the slots and propagates all of them to the `set` field of slot 0. The getSet function can then simply read the `set` of `announcements[0]` to get the current active set.

This algorithm is similar to the universal construction presented by Afek et al. in STOC'95 [2], and is *adaptive*; the step complexity of the insert and remove operations is proportional to the size of the active set plus the point contention during the insert operation in a given execution. This is because the number of slots that an insert operation traverses before finding one with no owner is at most the number of elements currently in the active set, plus the ones in the process of being inserted. We note that when using the active set object to count membership in a larger context (as was its original intent and is the way we use it for the lock algorithm), this translates to the point contention in the larger context.

5.2 Making a Multi Active Set

We now present an implementation of a multi active set that relies on an active set object. However, our multi active set object is not linearizable. Instead, we require a weaker property, which will suffice for our use of the multi active set object to implement locks. In particular, every multiInsert and multiRemove must appear to happen atomically at some point between the invocation and response. Any getSet operation that is invoked after that point, will see the effect of the operation, and any that responds before that point will not see the effect of the operation. However, any getSet that overlaps the point might or might-not see the effect. This property is remeniscent of *regularity* as defined for registers by Lamport [35]; we therefore call it *set regularity*.

We show how to implement a set-regular multi active set from a linearizable active set object in Algorithm 2. Each item is endowed with a flag that is initialized to false. To *multiInsert* an item into a given collection of sets, the item is first inserted into each of these sets using an active-set insert, and then its flag is set to true. The

```
1   struct Slot:
2   T owner
3   T* set
4   Slot[C] announcements

6   climb(int i):
7   for j = i ... 0:
8   for k = 1 ... 2:
9   curSet = announcements[j].set
10  if j == C: newSet = announcements[j].set //corner case
11  else: newSet = announcements[j+1].set
12  newMember = announcements[j].owner
13  if newMember != null:
14  newSet += newMember
15  CAS(announcements[j].set, curSet, newSet)

17  T* getSet():
18  return announcements[0].set

20  int insert(T p):
21  for i = 0... C-1:
22  if CAS(announcements[i].owner, null, p):
23  climb(i)
24  return i

26  remove(int i):
27  announcements[i].owner = null
28  climb(i)
```

Algorithm 1: Active Set Algorithm

```
1  type T:
2    void setFlag()
3    void clearFlag()
4    bool getFlag()

6  void multiInsert(T item, ActiveSet* collection):
7    item.clearFlag()
8    for set in collection: set.insert(item)
9    item.setFlag()

11 void multiRemove(T item, ActiveSet* collection):
12   item.clearFlag()
13   for set in collection: set.remove(item)

15 T* getSet(ActiveSet A):
16   T* set = A.getSet()
17   for T in set:
18     if not T.getFlag():
19       remove T from set
20   return set
```

Algorithm 2: Multi Active Set Algorithm

multiRemove operation first unsets the flag, and then removes the item from each of the sets. The getSet operation for the multi active set first calls the active set getSet operation, and then scans the items, returning the only ones for which it sees the flag is set to true. The flags can be scanned in any order, which implies the getset operation is not linearizable. For example, items a and b could be inserted into a set by two separate multiInserts, and for two getset operations that overlap the insert, one could return just a and the other just b.

We prove the following correctness and step complexity theorems in the full version of the paper [11].

Theorem 5.1. *The Multi Active Set algorithm presented in Algorithm 2 satisfies set regularity, assuming it uses a linearizable Active Set implementation.*

Theorem 5.2. *In Algorithm 2, each operation takes $O(\kappa)$ steps per active set it accesses.*

6 THE LOCK ALGORITHM

We now present the lock algorithm, whose pseudocode is shown in Algorithm 3. Intuitively, each lock is represented by an active set object that is part of a single multi active set object. Each tryLock attempt creates a *descriptor*, which specifies the list of locks to be acquired, the code to run if the locks are acquired successfully, and two other metadata fields: the *priority* assigned to this descriptor, and its current *status*. The status is set to `active` initially, and can be changed to `lost` or `won` later in the execution as the fate of this attempt is determined. The descriptor is used as the item to be inserted into the active sets; the priority field doubles as the flag for the multi active set; initially, it is set to -1, indicating a false flag.

After initializing its descriptor, a process starts its tryLock *attempt* with that descriptor. In a slight abuse of notation, we sometimes use a descriptor p to refer to the process that initialized p as it is executing the attempt tied to p. Without loss of generality we assume that each attempt is tied to a unique descriptor. At a high level, the algorithm implements each lock as an active set object, using Algorithm 1 (described in Section 5). A descriptor is inserted into the active sets of each of its locks via a multiInsert; to set the descriptor's flag to true in the multi active set algorithm, the negative priority is replaced with a uniformly randomly chosen value. The descriptor then calls `getSet` on each of its locks in turn, and compares its priority to that of all other descriptors in the set. Intuitively, if a descriptor p has the maximum priority of all descriptors on all of its locks, then it wins, and its thunk gets executed (celebrates).

However, the algorithm is more subtle, as it must block the adversary from skewing the distribution of a given descriptor p's competitors. Therefore, upon starting a new tryLock attempt, before calling the multiInsert, and in particular, before choosing a random priority, p *helps* all descriptors on its locks. Intuitively, this is done to 'clear the playing field' by ensuring that any descriptor whose priority might have affected p's adversarial start time cannot compete with p. To help other descriptors, p executes a `getSet` on each of its locks in turn, and, for each descriptor p' in the set, p helps p' determine whether it will win or lose. To do so, it calls the `run(p')` function, which serves as the helping function and is the way that a descriptor competes against other descriptors. We describe the `run` function in more detail below; this function is the core of the lock algorithm. Before describing it, we first explain what a descriptor p does after helping, and when it calls the `run` function to help itself.

After having executed the `run` function for every competitor, it is time for p to enter the game itself. First, p calls the multiInsert with its lock set as the argument. Recall that before returning, the multiInsert sets p's priority to a uniformly random value.[3] We call the time at which p's priority is written its *reveal step*, since it now reveals its priority to all other descriptors, and can now start receiving help from others. Note that by the set regularity property of the multi active set and the priority's use as p's flag in the multi active set, any `getSet` on one of those locks that starts its execution after p's reveal step will return a set that includes p. p now calls `run(p)` to compete in the game.

After returning from the `run(p)` call, p is guaranteed to have a non-active status (either `won` or `lost`). That is, it knows the outcome of its attempt. At this point, p cleans up after itself by calling multiRemove to remove itself from all active sets it was in.

The `run` *function.* The `run` function forms the core of the lock algorithm. The `run` function on a descriptor p checks the active sets of all of p's locks, and compares p's priority to all descriptors q in those sets such that q's status is `active`. On each such comparison, the descriptor with the lower priority is *eliminated*. This means that its status is atomically CASed from `active` to `lost`. After comparing p's priority with all descriptors in the active sets of all of its locks, the `run` function *decides* whether p won or lost. This involves trying to atomically CAS p's status to `won`. This will work if and only if p hasn't been previously eliminated. Finally, `run(p)` 'celebrates' the end of p's competitions by running its thunk if its status is `won`. The celebrationIfWon is also executed for each competitor that p faced. This ensures that any descriptor that reaches the `won` status gets its thunk executed before another descriptor sharing a lock wins, and ensures mutual exclusion.

Theorem 6.1. *Let κ be the maximum point contention any single lock can experience. Let L be the maximum number of locks in the lock set of any descriptor. Let T be the maximum length of a thunk. Algorithm 3 provides fine-grained locks such that the number of steps per tryLock attempt is $O(\kappa^2 L^2 T)$.*

Proof. It is easy to see this theorem holds by observing the tryLock and the multi active set algorithms. Each call to getSet takes an number of steps linear in κ, and each multiInsert and multiRemove takes $O(\kappa L)$ steps. Each instance of the `run` method calls getSet $O(L)$ times, and executes $O(T)$ steps for each descriptor in the resulting sets. Since the `run` method is called $O(\kappa L)$ times in a tryLock, this leads to the total step complexity of $O(\kappa^2 L^2 T)$. □

Delays. The algorithm as described thus far captures the essence of the approach; clear out any competitors whose priorities could have had an effect on your start time, and then compete by inserting yourself into the active sets of your locks and comparing your priority to all others. However, it also has weak points that the adversary can exploit to skew the priority distribution of the competitors of certain descriptors. In particular, a descriptor p takes a variable amount of its own steps to get to its reveal point, and a variable amount of steps after that to finish its attempt. This variance is caused by the amount of contention it experiences – how many descriptors are accessing the active set or are in it when p accesses the same active set, and what are their priorities. The number of other descriptors affect the time its insertion into the active sets takes (as shown in Section 5), as well as the number of descriptors it must compare its priority to. Furthermore, if p runs a descriptor's

[3] In this work, we assume that priorities do not conflict. To enforce this, it suffices to pick priorities in a range that is polynomial in the total number of processes, P, in the system. Conflicts can be handled by considering both processes to have lost, and would only slightly affect our bounds.

```
1  struct Descriptor:
2    ActiveSet* lockList //list of active set objects
3    thunk
4    int priority
5    status = {active, won, lost}

7    bool getFlag():
8      return (priority > 0)
9    void setFlag():
10     Delay until T_0 = c · κ^2 · L^2 · T total steps taken
11     priority = rand //reveal step of p
12   void clearFlag():
13     priority = -1

15 tryLocks(lockList, thunk):
16   p = new Descriptor(lockList, thunk, -1, active)
17   for each lock ℓ in p.lockList:
18     set = getSet(ℓ)
19     for each p' in set:
20         run(p')
21   multiInsert(p, p.lockList)
22   run(p)
23   multiRemove(p, p.lockList)
24   Delay until T_1 = c' · κ · L · T steps taken since previous delay

26 run(Descriptor p):
27   for each lock ℓ in p.lockList:
28     set = getSet(ℓ)
29     if (p.status == active):
30       for p' in set:
31         if (p'.status == active):
32           if p.priority > p'.priority:
33             eliminate(p')
34           else if (p != p'): eliminate(p)
35         celebrateIfWon(p')
36   decide(p)
37   celebrateIfWon(p)

39 decide(Descriptor p):
40   CAS(p.status, active, won)

42 eliminate(Descriptor p):
43   CAS(p.status, active, lost)

45 celebrateIfWon(Descriptor p):
46   if(p.status == won):
47     execute p.thunk
```

Algorithm 3: Lock Algorithm

thunk, this could take longer than if it simply eliminated it. The adversary can use this variance to skew the distribution of priorities of descriptors that p competes against.

To avoid this, we inject *delays* at two critical points in the algorithm. The first is immediately before p's reveal step. The goal is to ensure that p always takes a fixed number of steps from its start time until its reveal step. This means that once the adversary chooses to start p, it has also chosen its reveal time, and cannot modify this after discovering more information.[4] To achieve this goal, we choose a fixed number of steps until p's reveal step that is an upper bound on the amount of time p can take to arrive at its reveal step; $T_0 = c \cdot \kappa^2 L^2 \cdot T$, where κ is the maximum point contention on any lock, L is the maximum number of locks per tryLock attempt, T is the maximum number of steps to run a single thunk, and c is any any sufficiently large constant.

Similarly, we introduce a delay after p's run to ensure that the time between its reveal step and termination is also determined at its invocation. Here, there is no need to square κ and L, since p only needs to execute `run` for itself after its reveal step. Therefore, $T_1 = c \cdot \kappa LT$ for some sufficiently large constant c.

It is important to note that delay is in terms of steps for a particular process. The scheduler can run different processes at very different rates, so the delay counted in total number of steps across all processes in the history on one process could be very different than on another depending on the scheduler.

6.1 Safety and Fairness

We show that Algorithm 3 is correct by showing that it satisfies the mutual exclusion with idempotence property (Definition 4.3). The key to its correctness is in the way that the `run` function works. In particular, a descriptor's status can change at most once. Furthermore, `celebrateIfWon` never actually runs a thunk unless it status is `won` (at which point it cannot lose anymore). Since its status can become `won` only in Line 36, after it compares its priority to that of the descriptors on all of its locks, and also celebrates any winners out of these descriptors, by the time it celebrates for itself, the thunks of any earlier winners on any of its locks have already been executed. Thus, the placements of the celebrations (once on Line 35 for its competitors, and once on Line 37 for itself) are crucial for the safety of the algorithm. The full safety proof for the algorithm is presented in the full version of the paper.

We now focus on proving the fairness guarantees of the algorithm. In essence, we show that the adversary's power is quite limited. In particular, the adversary must decide whether or not two descriptors could threaten each other (i.e. their priorities could be compared in Line 32) before learning any information on either of their priorities. Intuitively, this is due to two main reasons.

The first is because of the helping mechanism. Before a descriptor p reveals its priority, it puts all descriptors whose priority was already revealed in a state in which they can no longer threaten it – their status becomes non-active.

To show this property formally, we begin with establishing some terminology; we say a descriptor p *causes a descriptor* p' *to fail* if p''s status is changed to `lost` in the `eliminate(p')` call on Line 33 or 34 after comparing p''s priority with p's priority. We say a descriptor p *can cause* p' *to fail* if p and p''s priorities are compared on Line 33 or 34 during the execution. We can now discuss when descriptors can and cannot cause each other to fail, in the next two useful lemmas.

Lemma 6.2. *A descriptor p cannot cause a descriptor p' to fail if their lock sets do not intersect.*

Proof. A descriptor p is only inserted into the sets corresponding to locks in its lock set (Line 21). Furthermore, in `run(p)`, only the sets of locks in p's lock sets are compared against. Therefore, a descriptor p' will never be compared against and potentially eliminated by a descriptor p if their lock sets do not intersect. □

Lemma 6.3. *A descriptor p cannot cause a descriptor p' to fail if p's status stopped being* `active` *before p''s reveal step.*

[4] For example, after p's start time, it's possible that some descriptor that started before p reaches its reveal time. At this point, the adversary has more information about p's competitors. It can attempt to extend p's time before its reveal step by starting new descriptors and forcing p to help them. We want to avoid this possibility.

Proof. First note that before p''s reveal step, no descriptor will eliminate p', since its priority will be negative, and the comparison with it will be skipped on Line 32. Therefore, p cannot cause p' to fail before p''s reveal step. By Lemma D.1, p's status will never be `active` again after it stops being `active`. Therefore, in any `run(p)` or `run(p')` call, the comparison of p with any other descriptor will be skipped on Line 29 after p becomes inactive. In particular, this comparison will always be skipped after p''s reveal step. □

Equipped with the above two lemmas, we can now show that the helping mechanism has the effect we want; descriptors whose intervals only overlap before one of their reveal steps cannot cause each other to fail.

Lemma 6.4. *Let p and p' be descriptors such that p's tryLock starts after p''s reveal step. Neither descriptor can cause the other to fail.*

Proof. By Lemma 6.2, if p and p''s lock sets do not overlap, the lemma holds. Otherwise, if p' has already removed itself from the active set by the time p started its `getSet` on Line 18, then p' must have already won or lost by this time. In particular, p wasn't in the active set during p''s `run(p')`, and therefore could not have caused p' to fail. Furthermore, by Lemma 6.3, p' cannot cause p to fail.

So, assume that p' is still in the active set at the time p started its `getSet` on Line 18. By the set regularity of the multi active set algorithm, since p' must have completed its insertion into the active set before p started its `getSet` on Line 18, p must have p' in the set it gets. Therefore, p calls `run(p')` before its own reveal step, so by Lemma D.2, p' wins or fails because of a different descriptor. Furthermore, again by Lemma 6.3, p' cannot cause p to fail, since a `run(p')` call completes before p's reveal step, and therefore p' is no longer active by that time. □

That is, in this lemma we show that if p starts after p''s reveal step, their priorities can never be compared. We can now introduce some more terminology to help us reason further about fairness. We define the *interval* of a descriptor p as the time between its calling process's call to tryLock and the time at which its tryLock call returns. Furthermore, a descriptor's *threat interval* is the time between the beginning of its interval and the time at which its status stops being `active`. We define p's *threateners* as the set of descriptors that can cause p to fail. We make the following observations about the relationships between descriptors.

Observation 6.5. *The set of descriptors whose intervals overlap a descriptor p's reveal step includes all of p's threateners.*

Proof. Every descriptor p's interval includes a complete call to `run(p)`. Thus, by Lemma D.2, the status of any descriptor is not `active` by the end of its interval. The lemma is therefore immediately implied from Lemmas 6.3 and 6.4. □

Observation 6.6. *At the time at which a descriptor's interval starts, none of its threateners have reached their reveal step.*

The other property of the algorithm which ensures that the adversary cannot pit descriptors against one another after knowing their priorities stems from the delays in the algorithm. In particular,

Observation 6.7. *Each descriptor interval takes the same number of steps by the initiating process between its start and its reveal step, and between its reveal step and the end of its interval, regardless of the schedule or randomness.*

Together, the helping mechanism and the delays allow us to prove the main lemma for the fairness the argument. This lemma relies on the notion of *potential threateners*. We say that a descriptor p is a *potential threatener* of another descriptor p' if (1) p's interval overlaps with p''s reveal step, and (2) p' did not execute a `run(p)`. Note that by Observation 6.5 and Lemma 6.3, the set of potential threateners of a descriptor is a superset of its actual threateners.

Lemma 6.8. *The player adversary has no information on the priorities of p or p' at the time at which it makes p' threaten p.*

Proof. By Observation 6.7, once a descriptor starts, its reveal step and last step of its interval are determined. Since these two steps are what determines whether a descriptor will be a potential threatener of another descriptor, the start times of the two descriptors determine whether this event occurs. Furthermore, by Lemma 6.4 and the definition of potential threateners, if p is a potential threatener of p' in an execution E, then both p and p' must have started their tryLock interval before either of their reveal steps. Therefore, the player adversary had no information on their priorities at the time at which it decided to start their intervals. □

The final theorem is easily implied from this lemma by recalling that the choice of priorities of each descriptor is always done uniformly at random and independently of the history so far. Thus, the adversary can choose whether or not to introduce more threateners for a descriptor p, but cannot affect their priorities. Since there is a bound on the amount of contention the adversary can introduce, we get a bound on p's chance of success.

Theorem 6.9. *Let k_ℓ be the bound on the maximum point contention possible on lock ℓ, and let $C_p = \sum_{\ell \in p.lockList} k_\ell$ be the sum of the bounds on the point contention across all locks in a descriptor p's lock list. Algorithm 3 provides wait-free fine-grained locks against an oblivious scheduler and an adaptive player such that the probability that p succeeds in its tryLock in A is at least $\frac{1}{C_p}$.*

Proof. On each lock ℓ in p's lock list, the adversary can make at most k_ℓ descriptors be potential threateners of p. Assume that all priorities of the descriptors are picked uniformly at random, but the priority of a given descriptor p' is hidden until after the adversary chooses whether or not p' will be a potential threatener of p. This is equivalent to our setting since the priorities are always picked uniformly at random, and, by Lemma 6.8, the adversary has no information on a descriptor p''s priority until after it decides whether it will potentially threaten p. Once the adversary discovers the priority of a descriptor, it can decide whether the next descriptor will be a potential threatener of p and then reveal the corresponding priority. In the worst case, the adversary can reveal up to C_p of those predetermined priorities. Since the set of potential threateners of p include all actual threateners of p, this makes p threatened by C_p uniformly chosen random values in the worst case, giving it a $\frac{1}{C_p}$ probability of having the maximum priority of all of them. □

Note that the theorem is stated using the point contention bounds of the specific locks that are in the lock set of tryLock attempt p. In terms of the general bounds κ on the point contention per lock and

L on the number of locks in any lock set, the probability of success can be bounded from below at $\frac{1}{\kappa L}$. Theorem 6.9 and Theorem 6.1 together imply Theorem 1.1.

Using the multi active set implementation. We note that as shown by Golab et al. [26], using implemented rather than atomic objects in a randomized algorithm can affect the probability distributions that an adversarial scheduler can produce. This effect can occur when several operations on the implemented objects are executed concurrently. Thus, we must be careful when using our set regular multi active set object in our randomized lock implementation. However, the way in which our lock algorithm uses the multi active set, and the way we use it in our analysis, is not subject to this effect. To see this, note that there is slack in our analysis; we consider the set of descriptors with *potential to compete*, where this means that a `getSet` executed by one descriptor *could* see the other descriptor. That is, any change in priority distributions that the adversary could try to achieve is already covered by our analysis.

6.2 Handling Unknown Bounds

So far, we've been assuming that κ, the upper bound on the point contention of any lock, and L, the upper bound on the number of locks per tryLock, are known to the lock algorithm. In this subsection, we briefly outline how to handle these bounds being known to the adversary but not the algorithm. The full algorithm and analysis for this case appear in the full version of this paper.

Algorithm 3 used these bounds in two ways: firstly, the active set objects were instantiated with arrays of size κ, and secondly, κ and L were used to determine how many delay steps each tryLock attempt must take to ensure that each descriptor's reveal step and final step of the attempt always happen after the same number of steps since the attempt's start time. The first concern is easy to fix by using more space; instead of setting the size of the announcement array of the active set object to κ, we set it to P, the total number of processes in the system. In most applications, this number is significantly larger than κ. We note that the size of each individual set pointed at by the array slots is still at most κ, and therefore the time bounds remain proportional to κ as well.

However, the second problem is more challenging, as the delays are crucial for the fairness bounds to hold. We make a few key changes to the algorithm. Firstly, must ensure that the size of the active set read on line 28 by an attempt p is fixed before the adversary discovers the priority of p. To do so, we split the reveal step into two parts; the *participation-reveal step*, and the *priority-reveal step*. The participation-reveal step occurs after a descriptor p inserts itself into the active set object of each of its locks. In this step, it changes its priority from -1 to a special `TBD` value, indicating that it is ready to participate in the lock competition, but not yet revealing its priority. At this point, all locks are queried to obtain their active sets, and only then is the priority of p revealed. The key insight is that after the priority is revealed, the active set objects are no longer queried, and instead the local copies of the sets, obtained just before the priority reveal step, are used. Thus, the adversary does not learn p's priority until after it can no longer affect the set of p's potential threateners.

However, there is still the issue of the steps a descriptor executes before its participation-reveal step. In the first part of its execution, a descriptor must help others to complete their `run` call, and must then call multiInsert on itself and its lock set. The length of these tasks vary depending on the number of active descriptors in the system, and can therefore be controlled by the player adversary. Instead of relying on κ and L, we employ a *doubling* trick; p measures the number of steps it took until right before its participation-reveal step, and then employs a delay to bring that number up to the nearest power of two. In this way, while the adversary still has control of the number of steps p will take, this number is now guaranteed to be one of only $\log(\kappa LT)$ values. We arrive at the following result.

Theorem 6.10. *Let k_ℓ be the bound on the maximum point contention possible on lock ℓ, κ be an upper bound on k_ℓ for all ℓ, and let $C_p = \sum_{\ell \in p.lockList} k_\ell$ be the sum of the bounds on the point contention across all locks in a descriptor p's lock list. Furthermore, let L be the maximum number of locks per tryLock attempt, and T be the maximum length of a critical section. Then there exists an algorithm A for wait-free fine-grained locks against an oblivious scheduler and an adaptive player such that (1) the probability that p succeeds in its tryLock is at least $\frac{1}{C_p \log(\kappa LT)}$ in A, and (2) A that does not know the bounds k_ℓ, κ and L.*

7 DISCUSSION

In this paper, we present an algorithm for randomized wait-free locks that guarantees each lock attempt terminates within $O(\kappa^2 L^2 T)$ steps and succeeds with probability $1/\kappa L$ where κ is an upper bound on the contention on each lock, L is an upper bound on the number of locks acquired in each lock attempt, and T is an upper bound on the number of steps in a critical section. We further show a version of the algorithm that does not require knowledge of κ and L, where the success probability is reduced by a factor of $O(\log(\kappa LT))$.

There are several interesting directions for further study. In particular, it is possible that a lock algorithm exists that reduces the number of steps per attempt to $O(\kappa LT)$. Furthermore, while the success probability of each attempt in our algorithm adapts to the true contention, our step bounds instead depend on the given upper bounds. It would be interesting to develop a lock algorithm that adapts its step complexity to the actual contention. To achieve such adaptiveness, an algorithm would necessarily have to avoid the delays that we use; our bounds rely on injecting fixed delays in which processes must stall if they finish an attempt 'too early'.

It would be interesting to see how well our proposed lock algorithm does in practice. It has recently been shown that lock-free locks can be practical [13], and we believe that the stronger bounds that our construction provides may be useful. In real systems, allowing the nesting of locks may be a useful primitive. While our construction allows acquiring multiple locks, these locks must be specified in advance and cannot be acquired from within a thunk (critical section). We believe that our algorithm would maintain safety if locks were nested, but its proven bounds would not hold. It would therefore be interesting to develop an algorithm for wait-free nested locks with strong bounds.

ACKNOWLEDGEMENTS

We thank the anonymous referees for their comments. This work was supported by the National Science Foundation grants CCF-1901381, CCF-1910030, and CCF-1919223.

REFERENCES

[1] Karl R. Abrahamson. On achieving consensus using a shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 291–302, 1988.

[2] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *ACM Symposium on Theory of Computing (STOC)*, pages 538–547, 1995.

[3] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling multi-object operations (extended abstract). In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1997.

[4] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 262–272. IEEE, 1999.

[5] Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014.

[6] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[7] Hagit Attiya and Eyal Dagan. Improved implementations of binary universal operations. *J. ACM*, 48(5), September 2001.

[8] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2006.

[9] Greg Barnes. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 261–270, 1993.

[10] R. Bayer and M. Schkolnick. *Concurrency of Operations on B-Trees*, page 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[11] Naama Ben-David and Guy E. Blelloch. Fast and fair lock-free locks. *CoRR*, abs/2108.04520, 2021.

[12] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

[13] Naama Ben-David, Guy E Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022.

[14] Guy E. Blelloch and Yuanhao Wei. LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS. In *International Symposium on Distributed Computing (DISC)*, 2020.

[15] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, page 257–268, 2010.

[16] Jeremy Brown, J. P. Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2002.

[17] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4):701–712, 1994.

[18] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1), 1986.

[19] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[20] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[21] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2014.

[22] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized dining philosophers without fairness assumption. In *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP*, pages 169–180. Kluwer, 2002.

[23] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5–es, 2007.

[24] George Giakkoupis and Philipp Woelfel. A tight rmr lower bound for randomized mutual exclusion. In *ACM Symposium on Theory of Computing (STOC)*, pages 983–1002, 2012.

[25] George Giakkoupis and Philipp Woelfel. Randomized mutual exclusion with constant amortized rmr complexity on the dsm. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 504–513. IEEE, 2014.

[26] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 2011.

[27] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[28] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2006.

[29] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[30] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[31] Peter Zilahy Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[32] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3), 1980.

[33] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981.

[34] Eyal Kushilevitz and Michael O Rabin. Randomized mutual exclusion algorithms revisited. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–283, 1992.

[35] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

[36] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 133–138, 1981.

[37] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proc. International Workshop on Data Management on New Hardware*, 2016.

[38] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *Uncertainty in Artificial Intelligence (UAI)*, 2010.

[39] Nancy A. Lynch, Isaac Saias, and Roberto Segala. Proving time bounds for randomized distributed algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 314–323, 1994.

[40] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[41] William Pugh. Concurrent maintenance of skip lists. Technical Report TR-CS-2222, Dept. of Computer Science, University of Maryland, College Park, 1989.

[42] Michael O. Rabin. N-process synchronization by $4 \log_2 n$-valued shared variables. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 407–410, 1980.

[43] Isaac Saias. Proving probabilistic correctness statements: the case of rabin's algorithm for mutual exclusion. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 263–274, 1992.

[44] Wattenhofer R. Schneider J. Bounds on contention management algorithms. In *International Symposium on Algorithms and Computation (ISAAC)*, 2009.

[45] G. Sharma and C Busch. A competitive analysis for balanced transactional memory workloads. *Algorithmica*, 4, 2012.

[46] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[47] Guy Lewis Steele Jr and Gerald Jay Sussman. Lambda: The ultimate imperative. Technical report, MIT CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1976.

[48] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Principles of Database Systems (PODS)*, pages 212–222, 1992.