Distributed Computing (2025) 38:51–72
https://doi.org/10.1007/s00446-024-00474-4

Fast and fair randomized wait-free locks

Naama Ben-David[1] · **Guy E. Blelloch**[2]

Received: 20 November 2022 / Accepted: 11 December 2024 / Published online: 24 January 2025

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 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 operation will take $O(\kappa^2 L^2 T)$ steps, regardless of whether it succeeds or fails. The operations are independent, thus if the tryLock is repeatedly retried on failure, it will succeed in $O(\kappa^3 L^3 T)$ expected steps. If the algorithm does not know the bounds κ and L, we present a variant that can guarantee a probability of at least $1/\kappa L \log(\kappa L T)$ of success. We assume an oblivious adversarial scheduler, which does not make decisions based on the operations, but can predetermine any schedule for the processes, which is unknown to our algorithm. Furthermore, to account for applications that change their future requests based on the results of previous tryLock operations, we strengthen the adversary by allowing decisions of the start times and lock sets of tryLock operations to be made adaptively, given the history of the execution so far.

Keywords Locks · Wait-freedom · Randomized algorithm

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, first introduced by Dijkstra, as a running example. At a high level, in the *dining philosophers problem*, 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. Many variants of the dining philosophers problem exist, varying in timing assumptions, as well as whether each philosopher knows its position around the table. We consider the *asynchronous, symmetric* setting, in which philosophers are symmetric and do not know their position around the table, and 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? Furthermore, 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

✉ Naama Ben-David
bendavidn@technion.ac.il

Guy E. Blelloch
guyb@cs.cmu.edu

[1] Technion, Haifa, Israel

[2] Carnegie Mellon University, Pittsburgh, PA, USA

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 [1–8], and of graph processing systems such as GraphLab [9]. 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 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 [10–12]. Turek, Shasha and Prakash [13] and Barnes [14], 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 [15].

2 Assumptions and approach

For the purpose of outlining the key assumptions, challenges and approach, and to generalize beyond dining philosophers

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

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

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 [16–19].

In our setting, in addition to the adversarial scheduler, the players can be adversarial, possibly trying to increase or decrease the probability of celebrating. When allowing processes to repeatedly attempt to acquire locks, it is unreasonable to assume that the players are oblivious, i.e., have 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 knowing 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 [17, 20–22], each player is assigned a *random priority* such that higher priorities win over lower priorities. In the synchronous, round-based setting, this by itself "solves" our problem. In particular, in each round, a set of players would play the game by (1) picking their priority, (2) checking if they have the highest priority in all their competitions, and (3) celebrating if they do. 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, 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 subtly, keeping the competition fair is challenging against an adaptive player adversary. Most obviously, if the player adversary wants a player to lose 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 gain knowledge by how the players are doing. Even an oblivious player adversary could skew the game; if players with high priority take more steps than ones with lower priority, incoming players would see a biased field of high priority 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 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). 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 several competitions at once, i.e., all the ones 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 adversary chooses the time at which a player p enters the game (in a pending state), as well as how quickly each player proceeds through the game. The adversary could therefore enter a new player p' while p is in a pending state, and have p' become active first. 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 [17] first considered the problem for a single lock, and only for the acquisition (i.e. no helping). Like ours, his scheme used priorities. Saias [23] showed the algorithm did not satisfy the claimed fairness bounds due to information leaks of the sort described in Sect. 2, and Kushilevitz and Rabin [24] fixed it with a more involved algorithm. Lehmann and Rabin also developed an algorithm for the dining philosophers problem [20]. Lynch, Saias and Segala [21] 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 [18, 19]. This work has focused on two classic models; the distributed shared memory (DSM) and the cache coherent (CC) models, both of which separate local from remote memory accesses in different ways. 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 [25] 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, even 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 [27].

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 Sect. 2—every philosopher 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 [28]. 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 [29], 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. [13] and independently Barnes [14] 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 [30] 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 [15]. The lock-free locks proposed in [15], 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 [31] 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 [32] 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 [30]). 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 [33–36], but it does not apply to the asynchronous setting we are considering.

Efficient wait-freedom. Starting with Herlihy [28], 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. [37] 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 [38] 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. [29] 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 [27].

4 Model and preliminaries

In this paper we use standard operations on shared 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 sequential 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 [25].

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 [39]. 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 [40] (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 shared memory via `Read`, `Write`, and `CAS`.

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 4.2.

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 [41] 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, $C_p = \sum_{\ell \in p.lockList} k_\ell$, where $p.lockList$ is the set of locks p attempts to acquire.

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 it was run once [10–12, 42]. 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 Sect. 5. We note that while for our use case, which is to use idempotent critical sections, CAS operations are not necessary, having a way to create idempotent code that includes concurrent operations opens up possibilities to use idempotence in other contexts. Barnes [14] and Turek et al. [13] do not extract the notion of idempotence, but do describe a way to make code based on reads and writes idempotent under our definition (below). However, they require that the reads and writes not access the same memory location concurrently.

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 $E|T$ 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.

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 4.2 (Mutual exclusion with idempotence) If a tryLock attempt p with thunk T and lock set L succeeds, then there is a run of T that executed to completion. Furthermore, T's interval does not overlap the interval of any other thunk whose lock set overlaps L. If p fails, there is no run of T.

5 Idempotence simulation

In this section, we describe a simulation that converts a thunk involving `Read`, `Write` and `CAS` instructions into one that is idempotent, and prove the following theorem.[3]

Theorem 5.1 *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*

[3] We note that a similar result is shown in [15], but the simulation technique shown here is different and we believe it is of independent interest. In particular, the technique of [15] is only suitable for short thunks, whereas ours does not have this restriction.

is idempotent, (2) every simulated memory operation takes constant time, (3) the simulated operations are linearizable.

We do this in a few steps. First, we present a construction with a few restrictions, and later, we show how to lift these restrictions. Namely, we begin by showing how to convert any thunk that (1) uses a constant amount of local memory, (2) does not make use of the return values of any CAS operation, and (3) we disallow concurrent Writes and CASes to the same location inside the thunk. To ease readability, we call CAS operations that do not return any value *compare-and-modify* (CAM) operations; thus, restriction (2) dictates that the thunk use CAM operations instead of CASes.

After presenting the idempotence construction for the restricted setting, we show how to lift these restrictions.

5.1 Idempotence construction for restricted thunks

In our simulation, each memory operation of the thunk is simulated by some number of other operations. To help us discuss concurrent memory operations, we split each simulated memory operation into an invocation and response, and refer to the operations used to implement the simulation as *primitive operations*. A simulation is linearizable if, for each memory operation *op* in the history of a simulation, the effect of *op* happens between *op*'s invocation and its response.

We base our simulation on two other operations, *labeled load link* (LLL) and *labeled store conditional* (LSC). An LLL takes a location and returns a "label" along with the value held at the memory location. An LSC takes a destination location loc, a label old, and a new value new. If old was the label returned by a LLL on loc, a later LSC on loc will *succeed* and write new to loc if and only if no successful LSC to loc linearized between the LLL and LSC. If old was not read from loc, the behavior is undefined. Note that the LLL and LSC are similar to standard load-link (LL) and store-conditional (SC) operations, but differ in that the LSC need not be executed by the same process as the LLL. Also here we are assuming the LSC does not return whether it succeeded or not, unlike usually assumed for SC.

The LLL and LSC operations can be implemented using standard techniques. For example, an implementation can attach a counter to every memory location and when executing an LLL, return the counter as the label along with the value. The LSC can then be implemented by first checking if the counter is the same and getting the old value, and if the same executing a double word CAM conditionally replacing the old value and old-counter, with the new value and counter incremented by one. This involves a double-word CAM (or CAS), but an implementation could avoid both a double-word CAS and an unbounded counter by instead using a level of indirection. In this case the label is actually a pointer to the value, and the CAM just has to use the pointer for its old, and a new pointer to the new value for its new. This requires some form of constant-time safe memory reclamation to ensure a pointer is only reused after no other process has access to it [43].

```
1  read(x):
2    return LLL(x).val

4  write(x, y):
5    lv = LLL(x)
6    LSC(x, lv.label, y)

8  CAM(x, old, new):
9    lv = LLL(x)
10   if (new != old and lv.val == old): LSC(x , lv.label, new)
```

Fig. 1 Simulating Read, Write and CAM using LLL and LSC

Our simulation works in two stages. Firstly, we implement traditional Reads, Writes and CAMs using LLL and LSC, as shown in Fig. 1. Secondly we simulate an LSC by adding a synchronization among processes. The idea of the synchronization is to have all processes working on a thunk agree on the state of the thunk before making any updates (LSCs). We define a *context* as the program counter and the current value of the registers for a thunk as it runs. A context can be used to represent the original thunk since the captured arguments (or pointers to them) can be kept in the registers, and the program counter would point to the first instruction of the thunk. We keep a copy of the context in shared memory, originally the initial thunk itself, and then periodically updated as the thunk proceeds. We refer to this as the *shared thunk*. Each process running a thunk also keeps its own copy of the most recent context it read from memory, which we call the *previous context*. We run the thunk as normal, except that before every LSC, we add a context update operation. The context update does a multiword (adjacent words) CAM on the shared thunk with the expected value being the previous context, and the new value being the current context. Note a multiword CAM can be implemented in constant time with a single-word CAM using indirection [43]. The purpose of the context update is to ensure that all runs of the thunk are in the same state before the LSC (what is important is that all LSC are applied with the same arguments). We refer to the instructions between two context updates as a capsule. The pseudocode is shown in Fig. 2. Without loss of generality we assume each context stored to the contextLocation is unique. If not, we can simply add a capsule number to the context to make it unique.

Lemma 5.2 *Any thunk using constant sized local memory and using only Read, Write and CAM operations on shared memory can be simulated using Reads, Writes and CAMs as primitive operations such that (1) it is idempotent, and (2) every simulated memory operation takes constant time. As long as there are no concurrent writes and CAMs on the same location, the simulation is linearizable.*

```
1  Local variables:
2    previousContext
3    sharedThunkLocation

5  run():
6    previousContext = Read(contextLocation)
7    load previousContext into registers
8    jump to program counter

10 runThunk(thunkPointer):
11   sharedThunkLocation = thunkPointer
12   run()

14 contextUpdate():
15    copy registers into newContext
16    CAM(sharedThunkLocation, previousContext, newContext)
17    run()

19 simulatedLSC(loc, old, new):
20   contextUpdate()
21   LSC(loc, old, new)
```

Fig. 2 Simulating code containing LLL and LSC so it is idempotent. The LLL is unmodified and the LSC uses the simulatedLSC

Proof First we outline the correctness (linearizability) of the implementation of concurrent Reads and Writes, and Read and CAMs in terms of LLL and LSC. We do not allow concurrent writes and CAMs. A Read will always linearize at its LLL. A Write will linearize on the LSC if successful, and otherwise linearize immediately before the write that corresponds to the first successful LSC after the LLL. There must be one before the Write's LSC since it is not successful. This gives the correct behavior when interleaved with Reads since a Write will either appear to be immediately overwritten by a following successful Write, or will be properly written after its LSC. The CAM will linearize at the LLL if either of the conditions fail, and at the LSC otherwise. Again this gives the correct behavior when interleaved with Reads.

We consider the simulation based on LLLs and LSCs. We consider how the thunk progresses through its capsules, proving correctness by induction on i, where i represents the i-th successful CAM of the context on Line 16 by any process helping with the thunk. In particular for i we assume that the thunk is idempotent to that point, and the context written is consistent with a single process running on the thunk to that point. We refer to the i-th context as c_i. At this point there might be processes that are still working on capsule i (the one before the i-th successful CAM) or even earlier capsules. Firstly we want to show that no LSCs from these earlier capsules can succeed. We note that if two calls are made to an LSC with equal arguments, the second always fails. This is because if the first LSC succeeds it falls between reading the label and the second causing the second to fail, and if the first fails some other LSC fell beween reading the label and it, which also falls between reading the label and the second, again causing the second to fail. Now any LSC from an earlier capsule $j \leq i$ will have read the same context as a capsule that has now completed its LSC, and therefore had the same arguments for its LSC, and will fail.

Secondly we take the first process running capsule $i + 1$ that executes its LSC step and add it to E' (from Definition 4.1). This is correct since it is the same instruction a single process running the thunk would take. Further LSCs on step $i + 1$ will fail since they have the same arguments.

Thirdly we want to ensure the next context written (at the end of capsule $i + 1$) is correct. This is because the first thing they all do is the LSC. Only the first can succeed, but immediately after the LSC they are all in the same state. This is where it is important that the LSC does not return whether it was successful or not. From this point until the end of the capsule the states might diverge since we are allowing for races with other procedures or thunks, but none of them are doing updates that are visible to each other, and at the end one will succeed on its CAM of a new context. That one is a valid run of just a single process for that capsule since the LSC, and we can add all its LLLs to E'. We have thus shown our invariants hold after the $i + 1$-th successful CAM of the context.

Finally when the first process finishes (responds), all previous LCSs in E' have been completed.

5.2 Removing the restrictions

Lemma 5.2 shows that any thunk that satisfies some assumptions can be made idempotent. In this section, we show that these assumptions are in fact not restricting; we can systematically convert any thunk that does not satisfy them into one that does. This will result in the final general result of Theorem 5.1.

We first handle the restriction on constant private memory. If a thunk uses more than a constant amount of private memory, we move its private memory into a shared memory region that is allocated specifcally for this thunk. It must then use shared reads and writes when accessing this memory instead of private ones. We make this change before applying the construction discussed in the previous subsection, and therefore, any reads and writes into this memory will be treated as shared ones and replaces with LLLs and LSCs as described in the construction. This adds constant overhead, and allows a thunk to use only shared memory.

Next, we note that Aghazadeh et al [44] present a construction that removes races between CAS and write operations by replacing the memory on which such races happens with a *writable CAS (wCAS)* object. This object is implemented in constant step complexity per operation using n^2 CAS primitive objects where n is the number of processes in the system. Ben-David et al [42] later presented a more space-efficient version. The idea of the construction is to use a level of indirection to direct CAS and writes on the same object to different memory locations. At a high level, the object stores

a pointer to the memory location in which its value its stored. Reads and CAS operations are executed on the value itself, after following the pointer to it. Write operations instead allocate a new memory location with the value, and then execute a CAS to swing the pointer to it. In this way, write and CAS operations that would have races on the same memory location no longer race. We can therefore use this construction to convert any thunk with write-CAS races into one in which writes and CASes do not race. The correctness of this construction (its linearizability and step complexity) is proven in [44].

Finally, we rely on another construction by Ben-David et al. [42] to show that any CAS can be implemented using only CAMs. That is, we show how to store the return value of a CAS separately from the CAS itself, to allow it to be recovered after executing the operation if an algorithm requires it. This construction uses only a constant number of CAM operations (i.e., does not require the return values of its primitive CAS operations). For this, we use the *recoverable CAS* construction presented in [42]. In their construction, they implement a CAS object with an extra 'recover' operation that can be called to recover the return value of the last CAS operation executed on the object. Their implemenation of this object uses a constant number of CAS primitive operations, and does not use the return value of any of them. We therefore use this construction directly to replace a CAS operation in a thunk by a CAM implementation as follows: call the CAS operation of the recoverable CAS object, discarding its return value, and then call the recover operation of that object. In this way, any thunk that makes use of its CAS operations' return value can be rewritten as a thunk with only CAMs. The correctness of this conversion is proven in [42].

In this way, we have removed all restrictions imposed by Lemma 5.2, and therefore Theorem 5.1 holds.

5.3 Practical considerations

Although the simulation is not particularly cumbersome, there is still overhead involved in the multiword CAM used in contextUpdate. Here we point out that there are several ways to reduce the number of contextUpdates (i.e. capsules). Importantly, any number of simulated LSCs on distinct locations can run in the same capsule, as long as they are all at the start of the capsule, i.e., immediately after the contextUpdate and before any LLLs. Also, any static data need not be written to the shared context. Many potential uses of our fair locks would require at most two capsules with only one or two registers saved across the boundary. This would be the case, for example, in locked operations on a stack or queue.

In follow-up work [15], we presented a different approach for an idempotent simulation, based on logging instead of context updates. We believe that the solution in the follow up work in more practical, but we note that to the best of our knowledge, the solution presented here is the first one that works in this general setting (i.e. for any thunk implemented with reads, writes, and compare-and-swaps).

6 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 Sect. 7, 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. [45]. 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 each process must alternate insert and remove calls.

The *multi active set* problem is a generalization of the active set problem to multiple sets. The object maintains a collection of sets, $\mathcal{S}$. Instead of an insert, a multi active set object supports a *multiInsert* operation, $multiInsert(e, S)$, that inserts an *element* e into a subset $S \subseteq \mathcal{S}$. The *multiRemove* operation, $multiRemove()$, operates with respect to the previous multiInsert operation by the same process, and removes the element from the sets it was inserted into. The getSet operation, $getSet(s)$, takes a set $s \in \mathcal{S}$ as an argument, and behaves the same as for the active set, returning all elements in the given set.

6.1 Active set algorithm

We now present a novel linearizable active set algorithm. Its pseudocode appears in Algorithm 3.

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. To do this in a linearizable manner, at every slot in the array, a process p executing the `climb` function attempts to update the set field twice, using a CAS. This ensures that even if it fails both times, the set gets updated by some process during the interval during which p executes the climb function, and therefore it still successfully collects the new owner of the slot before continuing to the next slot. 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 [37], 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.

```
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 = {} //corner case
11        else: newSet = copy(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)
```

We now prove Algorithm 3 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.

6.1.1 Correctness

We begin by showing that Algorithm 3 is a linearizable implementation of an active set object. Towards this goal, we first show some basic properties of the algorithm, which will then help us specify well-defined linearization points.

To ease the discussion of the algorithm's correctness, we introduce some helpful terminology. We assume that elements do not get reinserted into the active set after being removed. This is without loss of generality, as we can assign a unique id to each element when it is inserted. We say the owner of a slot i at time t is the element in `announcements[i].owner` at time t. Similarly, the *set* of slot i at time t is the set pointed at by the `announcements[i].set` field at time t. We say an element e is *current* at time t if there is an i such that e is the owner of slot i at time t. Note that at any time t, there may be elements in the `set` field of a slot that are not current. Furthermore, the current owner of a slot at time t may be null. We say that a process p *claimed* a slot i when its insert's CAS operation succeeds on slot i. We say that p *released* slot i when it sets i's owner to *null* in a call to `remove`. We let iteration ℓ of a call to the `climb` function be an iteration of the outer loop in the `climb` function where $j = \ell$.

Lemma 6.1 *At any point in time, the only current element that may appear in the set of a slot i but not in the set of slot $i+1$ is the owner of slot i.*

Proof We prove the lemma by induction on the number of times slot i's set changes. The lemma trivially holds at the beginning of the execution, since the initial state of the set of all slots is empty. Assume that the lemma holds after slot i's set changed at most k times. Consider the $k+1$th time it changes. Note that a set can only be changed by the CAS on Line 15 in the `climb` function. This CAS always puts in a new value that contains slot i's owner plus slot $i+1$'s set. Thus, the lemma still holds after this modification.

Observation 6.2 *The owner of slot i cannot change during the execution of `climb(i)` that started when i's owner was not null.*

Proof Note that the owner of a slot i only gets changed in an `insert` operation that claims i or in a `remove(i)` operation. At a given point in time, t, at which slot i's owner is not *null*, let p be the process that most recently claimed slot i. Recall by the specification of the active set object that p is the only process that can call `remove(i)` at time t. Furthermore, no other process can claim slot i until p calls `remove(i)`.

Lemma 6.3 *By the end of iteration i of a `climb` call, an owner of slot i that was current at some time after the beginning of this iteration is contained in slot i's set.*

Proof Note that if one of the two CAS instances called in iteration i of this `climb` call was successful, then the lemma holds. So, consider the case in which both CAS attempts of this iteration failed, and let q be the process that executed this call. Let R_1 be the first time that this `climb` call read the set of slot i on Line 9, and let R_2 be the second time. Furthermore, let C_1 and C_2 be the first and second time (respectively) that `climb` call executed a CAS on `announcements[i].set`. Note that between R_j and C_j, the algorithm reads the owner of slot i and adds it to the new value for the subsequent CAS. If both CAS instances failed, then slot i's set must have changed between R_1 and C_1, and again between R_2 and C_2. Note that slot i's set changes only during the execution of a `climb` through a successful CAS operation. Consider the process, p, that executed the successful CAS operation between R_2 and C_2. p must have read `announcements[i].set` after R_1, since otherwise its old value would have been outdated when it executed its CAS. Therefore, p must also have read `announcements[i].owner` after R_1, and added this owner to its new value for its CAS. Therefore, p's successful CAS placed an owner of slot i in slot i's set that was current at a time after R_1, and in particular, was current after the `climb` call of q began (note that the owner may have been *null*).

Lemma 6.4 *Let t_s be the time at which an instance of `climb(i)` is called, and let t_f be the time at which that instance of `climb(i)` terminates. There is a time $t' < t_f$ and an owner o such that (1) o was the current owner of slot i at some point between t_s and t' and (2) o is contained in the set of slot 0 at time t'.*

Proof To prove the lemma, we show that after iteration j of a `climb(i)` call, the owner of every slot k such that $j \leq k \leq i$ that was current at the beginning of the kth iteration or later is contained in the set of slot j.

We prove the lemma by induction on the iterations of `climb(i)`. Lemma 6.3 provides the base case; after iteration i, an owner of slot i that was current after the beginning of the call is contained in the set of slot i. Assume that after the kth iteration, an owner o of slot i that was current after the beginning of the call to `climb(i)` is contained in the set of slot k. If o is not *null*, by Lemma 6.1, slot k's set does not get changed to a set that does not contain o. Note that in the $k-1$'th iteration, two CASes copying k's set into $k-1$'s set are executed. Let C_1 and C_2 be these CASes, and let R_1 and R_2 be the corresponding reads of k's set on Line 9 that provided the old values of C_1 and C_2 respectively. If at least one of them is successful, then the lemma holds. Otherwise, there must have been a successful CAS by some other process that changed $k-1$'s set between C_1 and C_2. Note that slot $k-1$'s set is only ever changed during iteration $k-1$ of some `climb` instance. Therefore, the successful CAS must have copied k's set into $k-1$'s set. Furthermore, since this CAS succeeded after C_1 failed, it must have read its old value on Line 9 after R_1, and therefore copied an up-to-date value of k's set. Therefore the lemma holds.

We can now define the linearization points of the operations.

- The `getSet` operation linearizes at the read of the pointer to the set, on Line 18.
- The `insert(p)` operation linearizes at the first successful CAS on `announcements[0].set` that contained p in the new set value.
- The `remove(i)` operation linearizes at the first successful CAS on `announcements[0].set` after the remove operation began whose new value did not contain the descriptor erased from slot i by this `remove(i)`.

Note that since insert operations call `climb` on the slot at which they inserted an element, Lemma 6.4 and Observation 6.2 imply that there is a time during the interval of an insert operation at which the new element is present in `announcements[0].set`. Furthermore, recall that the set field is only ever modified with CAS operations. Therefore, the linearization point of an insert operation is well defined. Similarly, since a remove operation also calls `climb` on the index at which it removed an element, again by Lemma 6.4, there is a CAS operation during the remove operation's interval that changed `announcements[0].set` to a set that does not contain the removed element. Thus, the remove operation's linearization point is well defined as well. It is easy to see that these linearization points yield a correct sequential execution, since `getSet` reads `announcements[0].set`, and both the insert and the remove operations linearize when their effect becomes visible in `announcements[0].set`.

6.1.2 Step complexity

We begin by showing that the step complexity indeed depends on the point contention. We define the membership point contention at a time t, κ_t, to be the number of insert operations that have been invoked minus the number of remove operations that have linearized. We claim that the step complexity of the insert operations depends on κ_t where t is some point during the insert operation's interval.

Lemma 6.5 *If an insert operation's successful CAS is on `announcements[i].owner`, then at some point t during its execution, κ_t was at least $i+1$.*

Proof We prove this by induction on i. Clearly, if an insert successfully CASes its new element into slot 0's owner, then at the time at which it executed its CAS, there was at least

1 ongoing insertion – its own. Assume that the lemma holds for an insert operation that placed its new value in slot k. Consider an insert operation op that placed its new value in slot $k+1$. Consider the owners that op saw on slots $0-k$ that made its CAS fail. Call those owners the *critical owners*.

By the induction hypothesis, for every slot $j \leq k$, the insert operation of the critical owner of slot j had a point during its interval at which κ was at least j. Let that point for slot j be t_j. Consider the slot ℓ whose point t_ℓ is the latest of all such points. Then note that at t_ℓ, the insert operations of the critical owners of all slots $\ell \leq j \leq k$ must have already started, since otherwise their t_j points would have been after t_ℓ, contradicting the definition of t_ℓ. Furthermore, note that κ_ℓ must include at least $\ell+1$ elements that were not inserted into slots larger than ℓ, since we can build an execution that does not have those insertions and is indistinguishable to the process that inserted ℓ's critical owner. Therefore, t_ℓ is a point during which κ_{t_ℓ} was at least $k+1$. If t_ℓ is during op's interval, then we are done, since op itself adds another one to that total.

Otherwise, if t_ℓ is not in op's interval, then it must have been before op started, since all critical insertions were complete by the end of op's interval. In this case, all insertions of the critical owners that op encountered must have already started at the time at which op started, and since op saw these owners, their removals did not yet happen. Therefore, the invocation t_i of op's interval is a point at which $\kappa_{t_i} \geq k+2$. □

Note that the remove operation starts at the same slot as its corresponding insertion placed its element. Therefore, the remove operation traverses the same amount of slots as its corresponding insertion, which, by Lemma 6.5, is proportional to the membership point contention κ_t at some time t during the insertion. For simplicity, we use κ to refer to the maximum membership point contention during an insert operation, and use this to state our bounds in a more concise fashion. If the active set is used as a membership tracker wherein the insert and remove operations signify the beginning and end of an operation in some external system (as it is used in our lock algorithm), the insert operation's membership point contention is also a valid point contention for the overall operation in the external system.

We now briefly discuss the step complexity of the rest of the active set algorithm. Of interest is the `climb` function, which must make copies of the `set` fields of each slot. If we assume that it always makes deep copies, then we can consider each such copy to take time linear in the size of the set being copied. Since the sets never contain more than κ_t elements at any time t, we can bound this time by the point contention as well. Therefore, the insert and remove operations take $O(\kappa^2)$ steps. However, we can improve the time it takes to copy each set into the new one by implementing each set as a linked list, and *cons*ing the new element into the front of the list. More precisely, the `set` field of a slot i points to the head of the linked list representing its set. When executing the $i-1$th iteration of a `climb`, a process creates a new node containing the owner of $i-1$ and makes it point to the head of slot i's list. It then swings slot $i-1$'s `set` pointer to point to the newly added head. Notice that in this approach, the set of slot i is not changed in iteration $i-1$, since the linked list starting at the pointer from slot i's `set` field still does not contain the owner of $i-1$. Using this technique for maintaining and updating the sets, each iteration of the `climb` function takes constant time, and therefore, by Lemma 6.5, the total time for an insert or remove operation is $O(\kappa)$.

The `getSet` takes constant time, as it simply returns a pointer to the set. Note that this is still an atomic read of the set, since the set itself is never changed; for any i, every time `announcements[i].set` is changed, its pointer is simply swung to a new location.

Space Complexity. The `announcements` array has C slots, each of which can point to a set of at most C elements. However, these C elements are shared among the sets. In particular, each element only has one copy. Still, garbage collection must by employed to ensure the safe release of elements that have been removed from the set. This can be done using the wait-free reference counting presented in [46], yielding a total space usage of $O(C^2)$. If there are many active set objects in the same system (as is the case in our lock algorithm) the total space is C per active set object in the system plus $O(C^2)$.

6.2 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 [47]; 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 4. 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 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.

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

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

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

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

We now prove the correctness and step complexity bounds for the multi active set algorithm.

Theorem 6.6 *The Multi Active Set algorithm presented in Algorithm 4 satisfies set regularity, assuming it uses a linearizable Active Set implementation.*

Proof Let the linearization point of a multiInsert operation be at its setFlag. Let the linearization point of a multiRemove be at its clearFlag. We need to show that (1) any getSet of set S that starts before the linearization of an multiInsert(I, C) where $S \in C$ will not include I in its return set, (2) any getSet of set S that starts after the linearization of an multiInsert(I, C) where $S \in C$ and returns before the linearization point of the corresponding multiRemove returns a set that includes I, and (3) any getSet of S that begins after the linearization of the multiRemove will not include I in its set. To see this, note that an item whose flag is false is not returned in a getSet. Thus, (1) and (3) hold since each item's flag is cleared at the beginning of a multiInsert, before the multiInsert's linearization, and is also cleared at the linearization of the corresponding multiRemove, and is not set to true after that point. Thus, a getSet that returns before the linearization of a multiInsert or begins after the linearization of a multiRemove cannot return the relevant item.

To see (2), note that an multiInsert calls setFlag after calling insert on each of the sets in the collection C. Thus, since the underlying active sets are linearizable, the insert into each set has linearized by this point. Furthermore, the only remove operations called on the underlying active sets are through the multiRemove operation, after it calls clearFlag. Thus, between the linearization of a multiInsert(I, C) and the linearization of the corresponding multiRemove, S.getSet on any $S \in C$ will return I, by the linearizability of the underlying active set. Furthermore, since the flag of I is set at the linearization point of the multiInsert and not cleared until the linearization point of the multiRemove, a getSet(S) will not remove I from the set returned by the multi active set.

Theorem 6.7 *In Algorithm 4, each operation O takes $O(\kappa)$ steps per active set it accesses, where κ is the maximum membership point contention of the individual active sets O accesses during O's interval.*

Proof Using Algorithm 3 as the underlying active set object, each insert and remove operation takes $O(\kappa)$ steps. Therefore, the theorem holds for multiInserts and multiRemoves, each of which simply calls insert or remove (respectively) on each active set it accesses. The getSet method accesses only one active set, and iterates over each returned element once. Since the active set's getSet method takes constant time and returns a set of size at most κ, the multi active set's getSet takes $O(\kappa)$ steps.

7 The lock algorithm

We now present the lock algorithm, whose pseudocode is shown in Algorithm 5. 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. Without loss of generality we assume that each attempt is tied to a unique descriptor. We note that at any point in time, each process has at most one descriptor that represents its current tryLock attempt, and each descriptor is associated with the process that created it. We therefore sometimes refer to a descriptor and its associated process interchangeably, and denote a descriptor as p to also refer to its associated process.

At a high level, the algorithm implements each lock as an active set object, using Algorithm 3 (described in Sect. 6). 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 (i.e., its associated process *celebrates*).

However, the algorithm is more subtle, as it must block the adversary from skewing the distribution of the priorities of a given descriptor's competitors. Therefore, upon starting a new tryLock attempt, before calling the multiInsert, and in particular, before choosing a random priority, a process 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 process 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.[4] 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 its descriptor 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 determines whether p won or lost by trying to atomically CAS p's status to `won` (in the `tryToWin` function). This CAS succeeds if and only if p hasn't been previously eliminated. Therefore, after this CAS, p's status is no longer `active`. Finally, `run(p)` 'celebrates' the end of p's competitions by running its thunk if its status is `won`. The `celebrateIfWon` 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.

[4] 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 each 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   tryToWin(p)
37   celebrateIfWon(p)

39 tryToWin(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
```

Theorem 7.1 *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 5 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. By Theorem 6.7, each call to getSet takes a 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 their priorities are. The number of other descriptors affect the time its insertion into the active sets takes (as shown in Sect. 6), as well as the number of descriptors it must compare its priority to. Furthermore, if p runs a descriptor's 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.[5] 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, at the end of its tryLock operation, 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.

7.1 Safety and fairness

7.1.1 Safety

We show that Algorithm 5 is correct by showing that it satisfies the mutual exclusion with idempotence property (Definition 4.2). 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.

Lemma 7.2 *A descriptor's status can change at most once during its execution.*

Proof Note that the status of a descriptor is only changed in the `eliminate` and `tryToWin` functions, both of which use a CAS with an old value of `active` and if successful, change the status away from `active`.

Lemma 7.3 *After a call to `run(p)`, descriptor p is not in the `active` status and if it is in the `won` status, its thunk has been run.*

Proof The `tryToWin(p)` on Line 36 changes p's status away from `active`. By Lemma 7.2, once its status changes, it never changes again. Furthermore, `celebrateIfWon(p)` is called afterwards on Line 37, which, if p's status is `won`, runs p's thunk.

Lemma 7.4 *If p is in the `lost` state, its thunk is never run. Furthermore, a call to `execute p.thunk` only happens after p is in the `won` status.*

Proof A thunk is run only in a call to `celebrateIfWon`. The `celebrateIfWon` function does not run a thunk if the descriptor's status is not `won`.

Lemma 7.5 *If two calls `execute p.thunk` and `execute q.thunk` are executed concurrently, neither descriptor had its thunk run to completion earlier, and the lock sets of p and q overlap, then $p = q$.*

Proof Note that a descriptor's status can only change to `won` in the call to `tryToWin(p)` on Line 36 within a `run(p)` call, and that `run(p)` is only ever called after p's priority is set to a non-negative value.

Assume by contradiction that $p \neq q$, and let ℓ be a lock in the intersection of $p.lockSet$ and $q.lockSet$. By Lemma 7.4, both p and q must have reached the `won` status. Note that if p is in q's set on Line 18, then q must celebrate for p before q's reveal step, and in particular, `execute p.thunk` will execute to completion before `execute q.thunk` is called, contradicting the assumption that q and p are celebrated concurrently. For the same reason, q cannot be in p's set on Line 18.

[5] 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.

Without loss of generality, assume that p reaches its reveal step before q does. Consider the first instance of `run(q)` which executes `tryToWin(q)`. Let that instance be R, and let the time at which it calls `tryToWin(q)` be q_w. Note that any celebration for q must have occurred after q_w. Therefore, R must have had p in its set in Line 28, since otherwise this means that p was already removed from its set of locks, contradicting the assumption that p and q celebrated concurrently. Furthermore, R must have called `celebrateIfWon(p)` and executed it to completion before q_w, since q's status was `active` before q_w. Therefore, p was celebrated to completion before q's celebration started, contradicting the assumption that they celebrated concurrently.

Therefore, the mutual exclusion property is maintained, and a thunk is run if and only if a descriptor reaches the `won` status. In this case, we say that the descriptor succeeds in its tryLock attempt.

Theorem 7.6 *Algortihm 5 satisfies the mutual exclusion with idempotence property (Definition 4.2). Furthermore, a thunk is executed if and only if its corresponding tryLock attempt succeeds.*

7.1.2 Fairness

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 helping mechanism and the delays.

The helping mechanim. 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 by 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 (in a `run(p)` or `run(p')` instance). We define p's *threateners* as the set of descriptors that can cause p to fail. If p' is a threatener of p, we say that p' *threatens* p. More precisely, p' *threatens* p if p and p''s priorities are compared on Line 32 during the execution. Note that the *threatener* relationship is symmetric (i.e., if p is a threatener of p', then p' is a threatener of p), while the *caused to fail* relationship is not. We can now discuss when descriptors can and cannot cause each other to fail, in the next two useful lemmas.

Lemma 7.7 *Two descriptors p and p' do not threaten each other 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)`, (getSet) is only called on locks in p's lock set, so only descriptors that share some lock with p are compared against.

Lemma 7.8 *Two descriptors p and p' do not threaten each other if p's status stopped being `active` before p''s reveal step.*

Proof First note that before p''s reveal step, no descriptor can eliminate p', since it cannot appear in any lock's set. Therefore, p cannot cause p' to fail before p''s reveal step. By Lemma 7.2, 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 do not threaten each other if one of them starts its tryLock only after the reveal step of the other.

Lemma 7.9 *Let p and p' be descriptors such that p's tryLock starts after p''s reveal step. Then p and p' do not threaten each other.*

Proof By Lemma 7.7, if p and p''s lock sets do not overlap, the lemma holds. If p' is no longer `active` by the time p's `getSet` call on Line 18 returns, then by Lemma 7.8, the descriptors do not threaten each other. So, consider the case in which p' is still `active` at the time p's `getSet` call on Line 18 returns. Note that this also means that p' has not yet removed itself from its lock sets. Furthermore, by the set regularity of the multi active set algorithm and the fact that p''s reveal step was before p's tryLock started, 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 7.3, p' is no longer active by the time of p's reveal step. Thus, again by Lemma 7.8, the descriptors do not threaten each other.

That is, in this lemma we show that if p starts after p''s reveal step, their priorities can never be compared. 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.

We make the following observations about the relationships between descriptors.

Observation 7.10 *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 7.3, the status of any descriptor is not `active` by the end of its interval. The lemma is therefore immediately implied from Lemmas 7.8 and 7.9.

By the symmetry of the threaten relationship, we immediately have the following observation as well.

Observation 7.11 *At the time at which a descriptor's interval starts, none of its threateners have reached their reveal step.*

The delays Aside from the helping mechanism, 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 7.12 *Let O_1 be a tryLock attempt by process p_1. Let s_r be the number of steps p_1 takes executing O_1 until its reveal step, and s_t be the number of steps p_1 takes between O_1's reveal step and its termination. Let O_2 be any tryLock attempt by any process p_2. The number of steps p_2 takes until O_2's reveal step is s_r and the number of steps p_2 takes between O_2's reveal step and its termination is s_t.*

Together, the helping mechanism and the delays allow us to prove the main lemma for the fairness 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 7.10 and Lemma 7.8, the set of potential threateners of a descriptor is a superset of its actual threateners.

Lemma 7.13 *The player adversary has no information on the priorities of p or p' at the time at which it determines whether p' is a potential threatener of p.*

Proof By Observation 7.12, once a descriptor starts, its reveal step and last step of its interval are determined. Note that these steps are what determines whether a descriptor is a potential threatener of another descriptor, since a potential threatener's interval must overlap with the reveal step of the other descriptor. Therefore, the start times of the two descriptors determine whether this event occurs. Furthermore, by Lemma 7.9 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 7.14 *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 5 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 7.13, 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 7.14 and Theorem 7.1 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.

7.2 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. We present a modified version of the pseudocode in Algorithm 6. The lines in red are the ones that changed over the original version presented in Algorithm 5.

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

7    bool getFlag():
8      return (priority > 0 OR priority == TBD)
9    void setFlag():
10     Delay until total number of steps is the next power of two
11     priority = TBD //participation reveal step
12   void clearFlag():
13     priority = -1

15 tryLocks(Descriptor p):
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       if (p'.priority != TBD):
21         run(p')
22   multiInsert(p, p.lockList)
23   numContenders = run(p)
24   multiRemove(p, p.lockList)
25   Delay until numContenders · T steps taken since previous delay

27 int run(Descriptor p):
28   Lock[] sets
29   int numContenders = 0
30   for each lock ℓ in p.lockList
31     sets[ℓ] = getSet(ℓ)
32     numContenders += size(sets[ℓ])
33   CAS(p.priority, TBD, rand) //priority reveal step
34   for each lock ℓ in p.lockList
35     if (p.status == active):
36       for each p' in sets[ℓ]:
37         if (p'.status == active AND p'.priority != TBD):
38           if  p.priority > p'.priority:
39             eliminate(p')
40           else if (p != p'): eliminate(p)
41         celebrateIfWon(p')
42   tryToWin(p)
43   celebrateIfWon(p)
44   return numContenders

46 tryToWin(Descriptor p):
47   CAS(p.status, active, won)

49 eliminate(Descriptor p):
50   CAS(p.status, active, lost)

52 celebrateIfWon(Descriptor p):
53   if(p.status == won):
54     execute p.thunk
```

Algorithm 5 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 7.15 *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 does not know the bounds* k_ℓ, κ *and* L.

To prove this theorem, we follow a proof very similar to the version with known bounds.

Lemma 7.16 *Let* p *and* p' *be descriptors such that* p*'s tryLock starts after* p'*'s priority-reveal step. Then the descriptors do not threaten each other.*

The proof of this lemma is almost identical to the proof of its counterpart for the known-bound algorithm (Lemma 7.9), except that the reveal step is replaced with the priority-reveal step. We therefore omit it. We note that Lemmas 7.3, 7.7 and 7.8 apply to Algorithm 6 as well, so we can use them as-is in the proofs, with the reveal step replaced with the priority-reveal step.

We also make use of the following observation, which is a weaker form of Observation 7.12 that applies to the algorithm with unknown bounds.

Observation 7.17 *Let* O_1 *be a tryLock attempt by process* p_1. *Let* s_t *be the number of steps* p_1 *takes between* O_1*'s priority-reveal step and its termination.* s_t *is determined before* O_1*'s priority-reveal step.*

We can now prove the main theorem, using a similar argument to the proof of the fairness theorem of the known-bound version. The main difference is that here, a descriptor p's participation-reveal step is not completely predetermined at its start, but rather has $\log(\kappa LT)$ possible locations in the execution, where C_p is the maximum contention p can experience across its locks.

Proof of Theorem 7.15 At each of p's $\log(\kappa LT)$ possible participation-reveal times, the adversary can make at most C_p descriptors be potential threateners of p. Assume that the priority of each descriptor p' is picked uniformly at random at the beginning of the execution, but is hidden until after p''s priority-reveal time. This matches our setting since priorities are always picked uniformly at random at the priority-reveal time. By Observation 7.17, the number of steps that a descriptor p' takes after its priority-reveal time until it terminates is determined before p''s priority-reveal time. Furthermore, note that the priority-reveal time of each of a descriptor p's potential threateners must be after p starts, by Lemma 7.16. Additionally, by p's start time, p's participation-reveal time is fixed to be at one of $\log(\kappa LT)$ steps in the execution. Thus, before knowing p''s priority, the adversary must decide which of p's possible participation-reveal times p''s interval will overlap. Once the adversary decided this, it can reveal p''s priority, and based on this, can decide whether to reveal other descriptors' priorities. However, by the arguments above, there are at most $C_p \log(\kappa LT)$ potential threateners of p whose priorities it can reveal, and these priorities are all chosen uniformly at random. Therefore, p's chance of success is at least the chance that its uniformly random priority is the highest of $C_p \log(\kappa LT)$ uniformly random i.i.d. values; $\frac{1}{C_p \log(\kappa LT)}$.

8 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 we believe it may be possible to show that 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 [15], 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.

Funding Open access funding provided by Technion - Israel Institute of Technology. This work was funded by the National Science Foundation.

Declarations

Conflict of interest The authors have no other conflict of interest.

References

1. Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: ACM Symposium on Principles and Practice of Parallel Programming (PPOPP) (2014)
2. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 257–268 (2010)
3. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: ACM European Conference on Computer Systems (EuroSys) (2012)
4. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Conf. on Principles of Distributed Systems (OPODIS) (2006)
5. Kung, H.T., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Trans. Database Syst **5**(3), 354–382 (1980)
6. Bayer, R., Schkolnick, M.: Concurrency of Operations on B-Trees, pp. 129–139. Morgan Kaufmann Publishers Inc., San Francisco (1988)
7. Leis, V., Scheibner, F., Kemper, A., Neumann, T.: The art of practical synchronization. In: Proc. International Workshop on Data Management on New Hardware (2016)
8. Pugh, W.: Concurrent maintenance of skip lists. Technical Report TR-CS-2222, Dept. of Computer Science, University of Maryland, College Park (1989)
9. Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E., Hellerstein, J.: Graphlab: A new framework for parallel machine learning. Uncertainty in Artificial Intelligence (UAI) (2010)
10. Brown, J., Grossman, J.P., Knight, T.: A lightweight idempotent messaging protocol for faulty networks. In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2002)
11. de Kruijf, M.A., Sankaralingam, K., Jha, S.: Static analysis and compiler design for idempotent processing. In: ACM Conference on Programming Language Design and Implementation (PLDI) (2012)
12. de Kruijf, M., Sankaralingam, K.: Idempotent code generation: Implementation, analysis, and evaluation. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2013)
13. Turek, J., Shasha, D., Prakash, S.: Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In: Principles of Database Systems (PODS), pp. 212–222 (1992)
14. Barnes, G.: A method for implementing lock-free shared-data structures. In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 261–270 (1993)
15. Ben-David, N., Blelloch, G.E., Wei, Y.: Lock-free locks revisited. In: ACM Symposium on Principles and Practice of Parallel Programming (PPOPP) (2022)
16. Abrahamson, K.R.: On achieving consensus using a shared memory. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 291–302 (1988). https://doi.org/10.1145/62546.62594
17. Rabin, M.O.: N-process synchronization by $4 \log_2 n$-valued shared variables. In: IEEE Symposium on Foundations of Computer Science (FOCS), pp. 407–410 (1980). https://doi.org/10.1109/SFCS.1980.26
18. Giakkoupis, G., Woelfel, P.: A tight rmr lower bound for randomized mutual exclusion. In: ACM Symposium on Theory of Computing (STOC), pp. 983–1002 (2012)
19. Giakkoupis, G., Woelfel, P.: Randomized mutual exclusion with constant amortized rmr complexity on the dsm. In: IEEE Symposium on Foundations of Computer Science (FOCS), pp. 504–513 (2014). IEEE
20. Lehmann, D., Rabin, M.O.: 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), pp. 133–138 (1981). https://doi.org/10.1145/567532.567547
21. Lynch, N.A., Saias, I., Segala, R.: Proving time bounds for randomized distributed algorithms. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 314–323 (1994). https://doi.org/10.1145/197917.198117
22. Duflot, M., Fribourg, L., Picaronny, C.: Randomized dining philosophers without fairness assumption. In: Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP, pp. 169–180 (2002). https://doi.org/10.1007/978-0-387-35608-2_15
23. Saias, I.: Proving probabilistic correctness statements: the case of rabin's algorithm for mutual exclusion. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 263–274 (1992). https://doi.org/10.1145/135419.135466
24. Kushilevitz, E., Rabin, M.O.: Randomized mutual exclusion algorithms revisited. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 275–283 (1992)
25. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Prog Languages and Syst. (TOPLAS) **12**(3), 463–492 (1990)
26. Golab, W.M., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: ACM Symposium on Theory of Computing (STOC), pp. 373–382 (2011). https://doi.org/10.1145/1993636.1993687
27. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Inform. Control **70**(1), 32–53 (1986)
28. Herlihy, M.: Wait-free synchronization. ACM Trans. Prog. Languages and Syst. (TOPLAS) **13**(1), 124–149 (1991)
29. Afek, Y., Merritt, M., Taubenfeld, G., Touitou, D.: Disentangling multi-object operations (extended abstract). In: ACM Symposium on Principles of Distributed Computing (PODC) (1997)
30. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981)
31. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
32. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. (TOCS) **25**(2), 5 (2007)
33. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. In: ACM Symposium on Principles of Distributed Computing (PODC) (2006)
34. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: ACM Symposium on Principles of Distributed Computing (PODC) (2005)
35. Schneider J., W.R.: Bounds on contention management algorithms. In: International Symposium on Algorithms and Computation (ISAAC) (2009)
36. Sharma, G., Busch, C.: A competitive analysis for balanced transactional memory workloads. Algorithmica **4**, 299–322 (2012)

37. Afek, Y., Dauber, D., Touitou, D.: Wait-free made fast. In: ACM Symposium on Theory of Computing (STOC), pp. 538–547 (1995)
38. Attiya, H., Dagan, E.: Improved implementations of binary universal operations. J. ACM **48**(5), 1013–1037 (2001)
39. Ingerman, P.Z.: Thunks: a way of compiling procedure statements with some comments on procedure declarations. Commun. ACM **4**(1), 55–58 (1961)
40. Steele, G.L., Jr., Sussman, G.J.: Lambda: The ultimate imperative. Technical report, MIT CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB (1976)
41. Chor, B., Israeli, A., Li, M.: Wait-free consensus using asynchronous hardware. SIAM J. Comput. **23**(4), 701–712 (1994)
42. Ben-David, N., Blelloch, G.E., Friedman, M., Wei, Y.: Delay-free concurrency on faulty persistent memory. In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 253–264 (2019)
43. Blelloch, G.E., Wei, Y.: LL/SC and Atomic Copy: Constant Time. In: International Symposium on Distributed Computing (DISC), Space Efficient Implementations Using Only Pointer-Width CAS. In (2020)
44. Aghazadeh, Z., Golab, W., Woelfel, P.: Making objects writable. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 385–395 (2014)
45. Afek, Y., Stupp, G., Touitou, D.: Long-lived adaptive collect with applications. In: IEEE Symposium on Foundations of Computer Science (FOCS), pp. 262–272 (1999). IEEE
46. Anderson, D., Blelloch, G.E., Wei, Y.: Concurrent deferred reference counting with constant-time overhead. In: ACM Conference on Programming Language Design and Implementation (PLDI) (2021)
47. Lamport, L.: On interprocess communication. Distrib. Comput. **1**(2), 86–101 (1986)