The Distributed Computing Column

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

This month, the Distributed Computing Column is featuring Naama Ben-David, winner of the 2022 Principles of Distributed Computing Doctoral Dissertation Award. Her work on concurrent systems both builds critical theoretical foundations, while also addressing practical concerns of real-world systems. Underlying much of her work is a focus on performance: how do we design real concurrent systems that scale better and run faster? Within that context, Naama Ben-David has addressed a wide variety of important questions, such as the use of RDMA (remote direct memory access) memory, the impact of NVRAM (non-volatile random access memories), and how to design high-performance Byzantine agreement.

In this column, Naama Ben-David revisits the idea of "lock-free locks," an approach to concurrency that realizes many of the benefits of both lock-based and lock-free algorithms. Lock-free locks provide the same guarantee to a programmer as a typical blocking lock, while at the same time allowing for stronger progress guarantees, e.g., lock-freedom and wait-freedom. (This seemingly impossible combination is made possible by requiring that the critical section have the property of "idempotence," meaning that it can safely be executed more than once.)

Naame Ben-David gives an overview of the state-of-the-art for lock-free locks, and discusses several interesting open questions that remain. It is clear both that the approach is quite promising, and at the same time there is much work left to do!

The Distributed Computing Column is particularly interested in contributions that propose interesting new directions and summarize important open problems in areas of interest. If you would like to write such a column, please contact me.

LOCK-FREE LOCKS

Naama Ben-David
(VMware Research)

1 Locks

Modern systems make use of multiple processes to speed up tasks that can be parallelized. However, inevitably, when multiple processes run simultaneously in the same system and are accessing the same resources, they sometimes need to synchronize. More specifically, on modern multicore architectures, processes must coordinate accesses to the same shared memory to avoid overwriting each other's work and causing inconsistencies. This is the main challenge addressed in the study of concurrent programs; how do we ensure safe coordination among processes?

Perhaps the most common way to do this is through the use of *locks*. A lock is a primitive that protects a prespecified section of memory, and allows only one process to access that memory at a time. This allows processes to safely modify that memory without worrying about potential interference from other processes. The problem solved by locks is called *mutual exclusion*, first introduced by Dijkstra [19, 20]. In its simplest form, mutual exclusion specifies three sections of code that a process might be in; the entry section, the critical section, and the exit section. Intuitively, processes in the entry section are competing to acquire the lock, a process in the critical section is holding the lock, and processes in the exit section have just released the lock. Mutual exclusion guarantees that at any point in time, at most one process can be in the critical section, and that if there are processes in the entry section, eventually there will be a process in the critical section.

Since its introduction in the 1960s, the mutual exclusion problem has unsurprisingly garnered a lot of attention, with a lot of research into how to design mutual exclusion algorithms with better guarantees [36, 46], and fitting the requirements of new architectures [14, 18, 26, 32, 33], as well as several surveys on the topic [13, 45].

Use of locks in practice. Locks are used in many practical systems, including transactional systems [50], file systems [31], databases [16], and concurrent data

structures [8] to allow for increased parallelism without risking the safety of program logic. When designing a system or data structure using locks, an important decision must be made: at what *granularity* should the locks be used? In other words, how much memory should a single lock protect? This decision exposes a difficult tradeoff; it is simplest to write code when locks are coarse grained, meaning that each lock protects a large portion of the application's memory, since this usually means that only one lock must be acquired per operation. However, the more memory a single lock protects, the more likely it is that other processes will *contend* on that lock, therefore causing more sequential bottlenecks. Often, systems opt to use locks in a *fine-grained manner*. That is, rather than having a single global lock that protects the entire system, many locks are defined, protecting small pieces of memory. For example, in many transactional systems, one lock is assigned per data item [35,50,52]. This means that when executing a transaction on several data items at a time, the locks for all of them must be acquired before any change is made on any of the memory.

Downside of locks. While locks provide a simple abstraction for safely synchronizing concurrent processes, they suffer from a major drawback: when one process holds the lock, it blocks all others from accessing the memory that the lock protects. This may sound inevitable, since after all, preventing concurrent execution on that piece of memory is the goal. However, the manner in which it is done can in fact be quite problematic in practice, because for many different reasons, processes in a system often operate at very different speeds. For example, a process may be scheduled out by the system for a long period of time, during which it does not execute any code for the program, while others continue their execution. Another reason for stalling is caching issues, or architectural features that place some processes further away from parts of the memory than others. With all of these factors coming into play, bottlenecks can often form when a slow process is holding and not releasing a lock. This is especially when there is high contention, since then many other processes are waiting for this process to finish, causing a lot of wasted CPU cycles.

2 Lock-Freedom

Lock-free algorithms avoid this drawback of locks; they guarantee that progress is made in the system even if some process fails or stalls for an arbitrarily long time. Lock-free algorithms achieve this by carefully reasoning about the semantics of a program or data structure, and designing algorithms that can use just small atomic primitives that are provided by the hardware, like compare-and-swap (CAS), to synchronize processes. Lock-free data algorithms have been the topic of extensive

study, and many efficient lock-free data structures have been designed, including BSTs [9,11,21,43], queues [34,41,42], hash tables [40,44,48], priority queues [2, 7, 37, 49, 55], and linked-lists [28, 47]. However, lock-freedom comes at the cost of increased programming effort; the elegant abstraction that locks provide, which allows programmers to write sequential code and be guaranteed that it will be safe in a concurrent setting is no longer available.[1]

Various research efforts have been made to ease the design and implementation of lock-free programs. One approach has been to observe that many lock-free algorithms have a similar structure, and exploit that structure to extend and optimize many lock-free programs in a general way. For example, this was done in the definition of the *normalized form* of lock-free data structures [51], which was then used in several works to add useful properties to any normalized data structure [5, 15, 51]. In a similar vein, recent works have shown how to add range queries to a large class of lock-free search data structures [54], and how to make a different class of lock-free tree data structures persistent in an efficient manner [24]. Another approach to ease the design of lock-free algorithms has been to present helpful primitives that can be used instead of just individual word-sized CASes. For example, some work has introduced lock-free ways to extend CAS to affect two words at a time [25], or several words at a time [23, 27, 29, 38]. Similar primitives have also been introduced with a focus on making commonly recurring constructs in lock-free data structures simpler to implement [10, 12].

3 Lock-Free Locks: Best of Both Worlds?

So far, we discussed two approaches to synchronizing concurrent processes; locks, which are easy to use but can cause processes to *block* others from making progress, and lock-freedom, which does not block, but requires careful design and is difficult to generalize. We briefly surveyed a few efforts to make lock-free code easier to program through the design of useful lock-free primitives.

Lock-free locks offer an abstraction that achieves the best of both worlds, and can perhaps be seen as the most general extension of the above-mentioned trend of presenting easy-to-use lock-free primitives. More precisely, lock-free locks give the same interface and safety guarantee as a regular blocking lock (namely, that at any given time, at most one process holds the lock and that process can execute a critical section of code atomically on the protected memory), but provide a lock-free progress guarantee. At a high level, this is achieved by having a process holding a lock leave a *descriptor* of its critical section for others to see. Other

[1]While there are some *universal constructions* that can be used to generate general-purpose lock-free data structures [1, 22, 30], these constructions sequentialize all accesses to the data structure, and are thus inefficient.

processes contending on a lock then *execute the critical section of the current lock holder*. Once this is done, the contending processes can safely release the lock from the ownership of the previous process and take it for themselves.

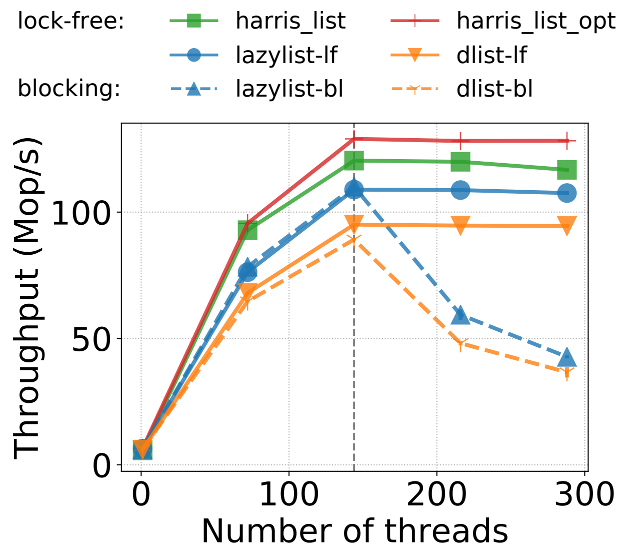
However, this must be done carefully, since having several processes potentially execute the same code simultaneously may result in that code having a different effect than intended, thereby breaking safety guarantees. Thus, for lock-free locks to work, the critical section that is run must be *idempotent*, i.e. it must have the same effect whether it is run just once or several times concurrently.

Origins of Lock-Free Locks. The idea of lock-free locks was first introduced in the 1990s by Barnes [3] and independently by Turek et al [53]. Both papers present similar ways of taking a fairly general class of code and converting it to be idempotent, thereby allowing lock-free locks to be used for any critical sections that fall within that class. However, the idempotent constructions presented in these early works required inefficient mechanisms that made them impractical. In particular, these papers achieved idempotence through a *context-saving* approach, in which after every instruction in the critical section, the entire context of the program (including program counters and registers) must be saved to allow others to continue the execution from that point. This heavy-handed approach would not only be slow, but would also require a special-purpose compiler to implement in practice. Lock-free locks have therefore been written off as impractical, and not been studied much further for about 30 years.

Renewed Interest. Recent work [6] has renewed interest in lock-free locks by introducing a new approach that makes them much more practical. At a high level, this new approach converts any critical section to an idempotent version of itself through a *log-based* mechanism rather than a context-saving one. This mechanism introduces a log that is shared among all helpers of a given critical section, and ensures that they all observe the same shared memory values for each instruction in the code by logging the first value observed by any process that ran the code. This approach does away with inefficient context-saving, instead requiring only one extra log access per instruction. Additionally, it allows the approach to be implemented in a simple library, thus enabling code written for blocking locks to be easily converted to its lock-free counterpart. In the next section, we describe this idempotence construction in more detail.

Practical potential. Ben-David et al. implemented their idempotence construction and the resulting lock-free lock abstraction in a library, demonstrating the potential that lock-free locks have [6]. In particular, they converted several lock-based data structures into lock-free ones by simply applying the library, and ran

experiments in various environments comparing the performance of lock-based data structures to that of lock-free ones. While their experiments were much more thorough, we show a representative plot from their paper here, in Figure 1.



(a) 100 keys, 5% up., $\alpha = 0.75$

Figure 1: Throughput of singly and doubly linked lists. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of the lock algorithm of [6], respectively.

The figure shows the empirical performance of various implementations of linked-lists (both singly and doubly linked). In particular, two hand-tuned lock-free implementations from the literature are shown (Harris's lock-free singly linked list [28] (`harris_list`), and an optimized version of Harris's list [17] (`harris_list_opt`)). Furthermore, two simple lock-based linked-list algorithms, one singly linked (`lazylist`) and the other doubly linked (`dlist`) are implemented, and converted to be lock-free using the lock-free lock library of [6]. Both their blocking and their lock-free version are shown. In the plot, blocking algorithms are represented with a dotted line. The plot shows the algorithms' scalability as the number of threads increases. A workload of 5% updates, split evenly between inserts and deletes, and 95% lookups is run, and keys are chosen according to a zipfian distribution with parameter 0.75. The experiment was run on a machine with 72 physical cores, each with two-way hyperthreading, so the number of parallel processes possible at any one time on the machine is 144.

Unsurprisingly, none of the algorithms scale much once the physical limit of parallelism is hit. However, it is interesting to note that the blocking algorithms' performance drastically degrades once that limit is hit, and the system becomes

oversubscribed (i.e., uses more processes than are available on the machine). This nicely demonstrates the downside of blocking locks; in an oversubscribed system, it is much more likely that a process holding a lock will be scheduled out by the system and will stall for long periods of time. The plot clearly shows that lock-free locks fix this problem. Importantly, in practice oversubscription may be difficult to avoid, since most machines are used for running several independent applications at a time, and those applications are unaware of each other's resource usage.

It is also clear that while the two lock-free algorithms that employ the lock-free lock library scale similarly to their hand-designed lock-free competitors, and much better than their blocking counterparts, they are still slower than the hand-designed versions. This is also unsurprising. Using a general methodology almost always implies giving up possible optimizations. However, the performance of the library-based versions is still competitive, and shows the potential that lock-free locks have.

The rest of this article. It is clear from the above discussion that lock-free locks have the potential to make lock-free algorithms easy to design and implement efficiently. However, there is still plenty of room to improve both their practical performance and their theoretical guarantees. In the rest of this article, we overview the current state of the art for lock-free locks; we discuss the notion of idempotence in more detail and present the idempotence construction of [6], and then discuss an algorithm that makes lock-free locks guarantee the stronger *wait-free* progress in a scalable manner [4]. After presenting these algorithms, we conclude the article with a brief discussion of the many open directions left to explore in this space. However, before delving into what is known and unknown about lock-free locks in more detail, we first briefly discuss realistic expectations about what lock-free locks can offer, and what they cannot.

3.1 What Lock-Free Locks Aren't

While we believe that the idea of lock-free locks carries a lot of potential, we must also recognize the limitations of this approach. In particular, there are various causes for slowdowns and bottlenecks in concurrent systems that cannot be fixed by replacing locks with their lock-free counterparts, regardless of how efficient the lock-free lock constructions can get. We now briefly discuss two such potential bottlenecks, where fixing them can be crucial for the performance of a system, but the solution will not be found by delving deeper into the lock-free lock approach.

Buggy critical sections. In this article so far, we discussed some reasons that processes may be slow while executing their critical section, for example, if they get scheduled out by the operating system. However, another potential reason for slowdowns experienced by a process holding a lock is bugs; if the critical section code runs into an infinite loop, for example, that process may never release the lock. The lock-free approach discussed here does not address this issue. In fact, employing lock-free locks in this situation may make matters worse; instead of having one process stuck trying to execute a buggy critical section, we may have multiple processes stuck executing that same code when trying to help. To address this source of slowdowns in lock-based system, an entirely different approach must be taken. Indeed, entire fields are dedicated to testing, debugging, and verification of software. When applying lock-free locks more broadly in practice, it would be good to combine their use with methods that ensure the correctness of the critical sections being helped.

Speeding up critical sections. Note that the lock-free lock approach has several processes redundantly (though safely) executing the same code. While this redundancy might be negligible when critical sections are short, this repeated helping can cause a lot of wasted work (CPU cycles). In some applications, the critical section that a process may want to run can be lengthy and slow, even without any performance bugs. Ideally, if many processes are all spending cycles trying to execute that critical section, one may think that that combined effort could be used to speed up the code. However, that is not what lock-free locks do. Instead, the different helping processes are each executing the entire critical section independently, racing to complete it in its entirety. This may in fact *slow down* the critical section further, since the processes may interfere with each other's cache locality. To speed up the execution of a critical section with more processes, the critical section must be carefully analyzed to find and expose its potential parallelism; this direction, while it may be very beneficial in some applications, is not explored in the study of lock-free locks.

4 Idempotence

The notion of idempotence appears in many different fields, including in linear algebra, networking, and recently, persistent memory. In all these settings, the meaning of idempotence is always intuitively the same; an operation is idempotent if applying it multiple times has the same effect as applying it just once. For concurrent programs, the definition of idempotence is a little bit more involved, since it must account for not only applying an operation multiple times sequentially, but also for the possibility that several concurrent processes executed the

same operation at the same time. It is thus surprisingly non-trivial to define. Here we briefly describe the definition of concurrent idempotence presented recently in [4] and discuss the intuition behind it. We note that similar notions have been used in the past – including in the early works surrounding lock-free locks [3,53], in the definition of normalized lock-free algorithms [51], and for persistent memory constructions [5] – but were never explicitly defined as idempotence.

Definition. To capture concurrent idempotence, we must first understand what a concurrent execution, or *history*, can look like. We model concurrency through a sequence of *steps*, where each step is an instruction executed by some process. Each process executes a sequence of steps that is dictated by the code it is running, and the steps of different processes are interleaved to form a concurrent *history*. When we discuss idempotence, we must refer explicitly to the code that generated these steps, to be able to determine whether this code is idempotent. Below is the definition of concurrent idempotence taken from [4].

> A piece of code C generates a sequence of steps S_C, which can depend on the state of memory. A step $s \in S_C$ is said to be *a step for* C, regardless of which process executes it. A *run* of a piece of code C is the sequence of steps taken by *a single process* to execute or help execute C. The runs for C can be interleaved. An *instantiation* of a piece of code C is a subsequence of the steps for C in a history H, possibly from many different runs, such that those steps are consistent with a single run of C.
>
> **Definition 4.1** (Idempotence [4])**.** *A piece of code C is idempotent if in any valid history H, there exists a valid instantiation H' of C that is a (possibly empty) subsequence of all operations from runs of C in H, such that*
>
> 1. *if there is a finished run of C (response on C), then the last step of the first such finished run must be the end of H', and*
> 2. *all steps for C in any of its runs in H that are not in H' have no effect on the shared memory.*

Intuitively, this definition allows the possibility that many different processes are executing *runs* of code C, but there is some way to combine the runs of many different processes into a subsequence of steps that were possibly executed by different processes, but together look as if they form a single run. That single run, or *instantiation* of C is intuitively 'the one that counts', and all other steps taken for C have no effect.

Log-Based Construction. We now discuss the construction of [6], which takes any piece of code C implemented from reads, writes, compare-and-swap, and memory allocation/de-allocation instructions, and constructs a version of it which satisfies the above notion of idempotence.

Recall that to use idempotence for safe lock-free locks, a process p must make a descriptor available in which it specifies its critical section. The idempotence construction presented in [6] takes advantage of the fact that there is already a descriptor shared by all processes wanting to execute this critical section and adds to its a *log* that is shared as well. The log's length corresponds to the number of instructions in the critical section, that is, there is one entry in the log per instruction in the critical section code. When a process (either p or another process contending on the lock) executes p's critical section, it uses a compare-and-swap to try to write the result of the ith instruction of its critical section execution into the ith slot of p's log. If the CAS fails, this means that another process has already executed this instruction; the process adopts the result written in the log as its own result, and proceeds from there. Intuitively, this ensures that all processes executing p's critical section read the same values (either directly from memory if they were the first to do so, or from the log otherwise), and therefore, assuming the critical sections are deterministic and processes do not rely on their private register values when executing a critical section, also write the same values.[2] Thus, overall, all executions of p's critical section have exactly the same effect. This construction is quite general. It works not only for reads and writes, but also if the critical section includes CAS instructions, and memory allocations and de-allocations.

The overhead that is introduced by this approach is easy to analyze. For each instruction in the original critical section, an extra CAS on the log is introduced. Note that if the lock is not highly contended and there is just one process executing this critical section at any point in time, then this overhead is minimal, since the log is likely cached (or mostly cached) for that process. However, if there is contention, i.e., at least two processes competing for this lock and concurrently executing this critical section, then each log access may constitute an extra cache miss, as cache coherence may move the log from the cache of one process to the cache of the other. In this case, it is likely that each non-log shared-memory access also causes a cache miss for the same reason. Thus, the log accesses approximately double the cache misses incurred by the program when there is contention. However, we note that this may constitute far more cache misses than the original process would have incurred were it to run its code using a traditional blocking lock, since cache coherence would not be a major factor in that scenario. As the

[2] These assumptions are fairly general. The second can be guaranteed by having the initiating process write its private register values in the descriptor along with the critical section code, so that all helpers can use the same values.

experimental results of Ben-David et al. show, this cost can be non-negligible, but may still pay off if processes are likely to be stalled for other reasons [6].

5 Wait-Freedom

Regardless of how they achieve idempotence, all lock-free lock constructions we discussed so far operate in the same manner: each lock has a descriptor pointer, which is `null` when the lock is free. When a process p wants to acquire a given lock ℓ, it tries to swing ℓ's descriptor pointer from `null` to its own descriptor using a CAS. If it succeeds, then p has now acquired the lock. Otherwise, this means that some other process p' has acquired the lock. p then helps p' run its critical section, which the descriptor specifies, in an idempotent manner, and then tries again to swing ℓ's descriptor pointer to its own descriptor.

This simple approach guarantees lock-free progress; as long as some process wants to acquire lock ℓ, some process will acquire lock ℓ and complete its critical section. However, there is no guarantee that any one specific process will succeed in its own acquisition of the lock as long as others are contending on it. In particular, in the description above, when p tries again to swing the pointer to its own descriptor, it may fail because a new process p'' did so first. This could continue forever, leaving p to help others continuously but never make progress for itself.

This phenomenon is by no means unique to lock-free locks. Many lock-free algorithms exhibit similar behavior; while global progress is guaranteed for the system, no individual process is guarantees to make progress. A stronger notion of progress is *wait-freedom*. A wait-free algorithm guarantees progress for each individual process within a finite number of its own steps. There's another advantage to wait-freedom: it allows us to easily discuss the complexity of an algorithm in terms of the number of steps that a process must take in the worst case to execute its operation. This is much more difficult to do in algorithms that are lock-free but not wait-free, since that number may be infinite.

This leads to a natural question: can we make lock-free locks have a *wait-free* progress guarantee? If so, how many steps does a process p need to take to acquire a lock?

First cut: a queue-based solution. One potential solution to this question would be to employ a wait-free queue per lock; processes contending on the lock can enqueue themselves onto the queue, and then help everyone ahead of them before acquiring the lock for themselves. A similar approach is commonly used in the implementation of *fair* solutions to the mutual exclusion problem (except that processes wait for their turn without helping in the case of traditional blocking locks) [32,39,46]. This solution could work quite well for lock-free locks as well.

The number of steps each process would take to acquire the lock in this case would be proportional to the contention it encountered (i.e., how many other processes were ahead of it in the queue) and the number of steps it takes to help each process, plus some overhead to enqueue itself at the beginning. If an efficient queue is used, this solution can be quite efficient.

However, there is another consideration to take into account. As discussed in Section 1, many practical lock-based systems employ locks at a fine granularity. In particular, this means that processes are likely to acquire not one lock, but several locks simultaneously before being able to execute their critical sections. In this setting, the queue-based approach can quickly lose its good step complexity guarantees. Consider, for example, a scenario in which each process in the system wants to acquire at most 2 locks, and each lock has at most 2 processes contending on it at any given time. This relatively low-contention setting should intuitively allow each process to complete its execution quickly, within a constant number of its own steps. However, long dependency chains can form; if a process p wants to acquire lock ℓ_1, which has process p_1 already on its queue, p must help p_1 complete its critical section first. If p_1 must first acquire lock ℓ_2 before it can execute its own critical section, then process p must help it acquire ℓ_2 as well. However, before doing so, it might need to help a process p_2 acquire ℓ_2, which may then need to acquire ℓ_3 as well. In this way, the number of processes that p must help before executing its own critical section could blow up to include the total number of processes in the system, despite only facing a small constant number of contenders on its own lock.

A randomized approach. Recent work has addressed this problem and presented a randomized protocol for wait-free locks in which the expected number of steps to acquire a set of locks does not depend on the size of the entire system [4]. In more detail, the algorithm considers *tryLock attempts* in which a process specifies a subset of the locks in the system that it wants to simultaneously acquire, and the critical section it wants to run if successful. A tryLock attempt may fail, in which case the locks are not acquired, the critical section is not run, and the process may try again in a new attempt. The algorithm guarantees that each attempt has a *fair* chance to succeed; if each attempt aims to acquire at most L locks, and each lock has at most C processes contending on it at any time, then the attempt has a chance of at least $1/CL$ to succeed. Furthermore, the number of steps a process takes per attempt is $O(L^2 \cdot C^2 \cdot T)$, where T is the maximum length of a critical section.

At a high level, the algorithm works by assigning a random priority to each contending process, and processes only help those who have a higher priority than their own. Each tryLock attempt gets a single priority that it uses for on all

its locks for this attempt. Each lock has a 'competing set' that can be thought of as the equivalent of its queue in the queue-based approach; this set keeps track of the processes currently contending on the lock. When starting an attempt, a process p does the following: (1) it adds its descriptor to the competing set of each lock in its desired lock set, but without specifying its competing priority. (2) it checks all of the competitors on its competing set, helps the one with the highest priority on each lock, and 'kills' all the others. If some competitor doesn't have a priority, it is skipped (not helped and not killed). That is, for any attempt in the competing set of some lock, if its priority exists but is not the highest in this set, then p sets its state to 'aborted'. No aborted attempts will be helped in the future. (3) Now p finally chooses a random priority and updates its descriptor accordingly. Process p now repeats the second step, this time with the possibility that it will be the winner.

The random priorities help avoid the long chains that prevented the queue-based approach from scaling; rather than having to help a process until it succeeds in its critical section, a process p helping a process p_1 may now abort p_1 if p_1 does not have the top priority on its other locks. This is the key idea that helps the randomized algorithm achieve its good step complexity bounds.

Subtleties and downsides of the algorithm. When analyzing randomized concurrent protocols, an *adversary* is used to model the system scheduler to capture worst case executions. It is generally assumed that the adversary does not know the future, so does not know the results of future coin flips or random decisions, but can know what has happened so far in the execution. This adversary can be quite powerful in skewing the probability that a process p will succeed. For example, in the algorithm of [4], p goes through a first round of helping before choosing its own priority and starting to compete to avoid effects that the adversarial scheduler could have. In particular, if p were to choose its priority and competes immediately, an adversary that wants to decrease p's probability of success can wait until it sees that p's competitors currently have relatively high priorities, and only let p compete at that point. This would inherently skew p's chances of success.

The algorithm of [4] achieves bounds that depend on the maximum number of locks requested in each attempt, the maximum amount of contention per lock, and the maximum length of a critical section. These bounds must be known in advance to the algorithm, as it in fact makes use of these bounds explicitly. In particular, it injects an artificial *delay* before choosing p's priority during the execution of an attempt, making p potentially take extra useless steps just to waste time, to ensure that p always takes the same number of steps between the beginning of its attempt and the point at which its priority is revealed to the adversary. This is done to

prevent the adversary from using the number of steps p takes during its execution to skew its probability of success. These delays are fairly unsatisfying, but are required to achieve the guaranteed bounds.

6 Open Problems

Since their introduction in the 1990s, lock-free locks have been mostly dismissed in the literature as impractical, and their study has only been renewed recently. This leaves our understanding of lock-free locks in its infancy, with many open problems, both theoretical and practical, left to be resolved. To wrap up this article, we briefly outline some of these promising future directions.

Reducing the overhead of idempotence. Lock-free locks must inherently introduce some overhead as compared to their blocking counterparts, since they must ensure that the code that is run in their critical sections is idempotent. The potential of lock-free locks was only understood when an idempotence construction with relatively low overhead was introduced last year [6]. However, it may be possible to improve this overhead, thereby immediately improving the performance and practicality of lock-free locks, as well as other applications of idempotence (see Section 4 for a brief discussion of such applications).

The goal of improving the overhead of idempotence constructions can be approached from several angles. The most immediate one from the discussion in this article is to find a construction that is as general as the one presented in [6], but more efficient. As discussed in Section 4, the construction of [6] already only introduces constant overhead in theory, making it difficult (though maybe not impossible) to improve the theoretical overhead. However, there may be plenty to do to improve its overhead in practice. In particular, this construction exhibits poor cache locality due to coherence issues when sharing the log among concurrent processes. It would be interesting to study whether an idempotence construction can be found that suffers less from coherence misses. Furthermore, we note that aside from the coherence issues, the construction of Ben-David et al. forces each new helping process to execute the entire critical section from the beginning, potentially wasting a lot of redundant work. This may be fine for short critical sections, but can become unacceptable when critical sections are long. Another direction for improving this construction's overhead is to find a way to allow helpers to skip ahead to where the most advanced process has reached in the critical section code. This is something that is achieved by the context-saving approaches discussed in Section 3, but at too great a cost. Is there a way to avoid having each process re-execute the entire critical section without resorting to saving the entire context of each process after each instruction?

On a similar vein, another way to optimize the construction would be to reduce its space overhead. The log used in the construction of Ben-David et al. is as long as the number of instructions in the critical section. This may be ok overhead for short critical sections, but may become prohibitive if critical sections are longer. Is there a way to reduce the space overhead required for idempotence?

A different approach to improving idempotence overhead can also be taken. Namely, rather than sticking with an idempotence construction that can be applied to extremely general code, one can ask whether there are some types of critical sections that are naturally more amenable to becoming idempotent with minimal overhead. Of course, an immediate answer is yes – some critical sections may be idempotent to begin with, therefore requiring no overhead at all. This opens up the possibility that one could define classes of code that are in the middle; not already idempotent as-is, but may easily become so. Understanding how general such classes of code may be could open up the potential to employ lock-free locks in various real-world applications almost for free. Interestingly, this direction ties in nicely with the very active research area on lock-free primitives and ways to make lock-freedom more easily applicable in general (see Section 2 for a brief discussion of these works), as such studies also aim to identify general types of code that are both useful in many applications and easy to make lock-free.

Improving theoretical guarantees of wait-free locks. We know that wait-free locks can be achieved in a fine-grained lock system in time proportional to the number of locks each process may acquire at once and the amount of contention each lock may have at any given time. These bounds are quite good for many systems, since they don't depend on the total number of locks in the system or the total number of processes in the system, both of which could be huge compared to the local contention or the size of each individual operation.

However, the bounds we have leave a lot to be desired. Firstly, it's possible that the dependencies on L, the maximum number of locks per attempt, and C, the maximum contention per lock, could be decreased. Currently, for a process to successfully acquire its locks, it needs $O(L^3 \cdot C^3 \cdot T)$ steps in expectation, with T being the maximum length of a critical section. Is it possible to reduce this to only a linear dependence? Perhaps more interestingly, recall that for these bounds to hold, L, C, and T must be known to the algorithm in advance. The algorithm loses its complexity guarantees if these bounds are surpassed at any point. This is somewhat unsatisfying; many systems may not know exact bounds on contention and size of operations in advance, and may experience workloads in which there are long periods of low contention and small operations, with intermittent busy periods where these figures are much higher. Finding an algorithm that can adapt to the *actual* amount of contention and size of operations in an execution is therefore

an important future direction.

Finally, we also note that the algorithm presented in [4] is randomized, and therefore all bounds are given in expectation rather than in the worst case. Recall that the queue-based approach described in Section 5 is deterministic, but its worst case step complexity bounds depend on the total number of processes in the system. Is it possible to find a *deterministic* wait-free lock algorithm that does not depend on the total size of the system?

References

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

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–20, 2015.

[3] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, 1993.

[4] Naama Ben-David and Guy E Blelloch. Fast and fair randomized wait-free locks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 187–197, 2022.

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

[6] Naama Ben-David, Guy E Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (SPAA)*, pages 278–293, 2022.

[7] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. Cbpq: High performance lock-free priority queue. In *European Conference on Parallel Processing*, pages 460–474. Springer, 2016.

[8] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Symposium on Principals and Practice of Parallel Programming*, 2010.

[9] Trevor Brown. A template for implementing fast lock-free trees using htm. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 293–302, 2017.

[10] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 13–22, 2013.

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 329–342, 2014.

[12] Trevor Brown, William Sigouin, and Dan Alistarh. Pathcas: an efficient middle ground for concurrent search data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 385–399, 2022.

[13] Peter A Buhr, David Dice, and Wim H Hesselink. High-performance n-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, 2015.

[14] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 157–166, 2013.

[15] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *ACM SIGPLAN Notices*, 50(10):260–279, 2015.

[16] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. {SplinterDB}: Closing the bandwidth gap for {NVMe}{Key-Value} stores. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 49–63, 2020.

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[18] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 247–256, 2012.

[19] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968.

[20] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[21] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.

[22] Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

[23] Steven Feldman, Pierre LaBorde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, 43(4):572–596, 2015.

[24] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 377–392, 2020.

[25] George Giakkoupis, Mehrdad Jafari Giv, and Philipp Woelfel. Efficient randomized dcas. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1221–1234, 2021.

[26] George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized rmr complexity on the cc model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 221–229, 2017.

[27] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In *International Symposium on Distributed Computing (DISC)*, 2020.

[28] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*.

[29] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing (DISC)*, pages 265–279. Springer, 2002.

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

[31] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. {BetrFS}: A {Right-Optimized}{Write-Optimized} file system. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, 2015.

[32] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Towards an ideal queue lock. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, pages 1–10, 2020.

[33] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 177–186, 2019.

[34] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 223–234, 2011.

[35] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[36] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 1974.

[37] Yujie Liu and Michael Spear. A lock-free, array-based priority queue. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 323–324, 2012.

[38] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 314–323, 2003.

[39] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 165–171. IEEE, 1994.

[40] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.

[41] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[42] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. Bq: A lock-free queue with batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 99–109, 2018.

[43] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

[44] Jesper Puge Nielsen and Sven Karlsson. A scalable lock-free hash table with open addressing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–2, 2016.

[45] Michel Raynal and Gadi Taubenfeld. A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47–65, 2022.

[46] Michael L Scott and William N Scherer. Scalable queue-based spin locks with time-out. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 44–52, 2001.

[47] Niloufar Shafiei. Non-blocking doubly-linked lists with good amortized complexity. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[48] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

[49] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[50] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[51] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 357–368, 2014.

[52] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[53] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222, 1992.

[54] Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 31–46, 2021.

[55] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–278, 2015.