An Overview of Lock-Free Approaches to Concurrent Data Structures & Algorithms

Alice Ao
Yale University

Abstract

This literature review explores lock-free data structures and algorithms as alternatives to traditional lock-based mechanisms. While systems and network developers have traditionally used locking mechanisms to guarantee safe concurrency, these mechanisms can incur significant performance costs. This review first discusses these performance costs as a motivation behind building lock-free data structures and algorithms. It then explores three illustrative examples of lock-free implementations that use common approaches of compare-and-swap and retry loops. The review addresses some challenges of lock-free programming and lastly provides a survey of the many domains that can be made lock-free, including real-world networks and communications systems.

1 Introduction

Concurrency is key to distributed computing, as processes from different computers and networks should be able to safely share memory and resources. Ideally, only one process should modify a section of code at a time, and other processes should then be able to read the modified value(s).

The traditional method of synchronizing shared memory relies on *mutual-exclusion locks* [5]. Typically, a mutual-exclusion lock protects a critical section of code. The process or task that currently holds the lock will have exclusive access to that section, preventing other processes from reading or writing to the section. While locks may have some optimizations, such as reader-writer locks that allow simultaneous reads of the same critical section, writes still require exclusive access. The locking approach to synchronization often leads to performance issues, as processes may need to unnecessarily wait on one another.

Therefore, this review seeks to explore lock-free alternatives for safely synchronizing shared memory. These alternatives can be used to construct concurrent data structures, such as linked lists and hash tables, and extended to practical applications, such as memory allocation algorithms and network communications.

2 Motivation

2.1 Performance Issues with Locks

While locks, when used properly, can prevent safety issues like multiple processes simultaneously accessing and overwriting the same memory, they can create performance bottlenecks [8]. When multiple processes may need to use and wait on the same lock, a process that currently controls a lock can delay, or even prevent, other processes from progressing.

With a *priority inversion*, a lower-priority process takes hold of a lock first and prevents other processes from accessing the lock's critical section simultaneously. Therefore, some higher-priority processes may need to wait for this low-priority process to release the critical section [1–3, 10]. This is especially detrimental if the lower-priority process takes an extremely long time to run, or if the lower-priority process is not scheduled to run for a while, preventing the higher-priority processes from finishing before their deadlines. The performance issues of priority inversions have had drastic real-world consequences, including frequent shutdowns of a Mars rover that hindered a Mars mission.

An even more severe problem with locking is *convoying*, in which the process that currently holds the lock is interrupted or de-scheduled. Since this process is unable to release the lock, other processes that require access to the critical section will be prevented from making progress [3].

Yet another possible problem is *deadlock*, in which processes attempt to grab the same locks, but in different orders. Since no process will release its current lock(s), which other processes are waiting on, they are in a deadlock, futilely waiting on one another to finish. Therefore, none of these processes can make progress [2, 3].

Lock-based techniques also require frequent use of lock variables, which unnecessarily increases the number of memory accesses. Some popular lock-based techniques may also require large processor-to-memory bandwidth. Spin-locking, for instance, involves repeatedly attempting to modify a value and testing to see whether the value has changed, which signifies that the lock has successfully been acquired [3].

Additionally, in practice, locks provide concurrency control that is significantly stricter than some systems may need. In the context of high-traffic networks like the Internet, two-phase locking is costly and largely unsuitable. Obtaining locks often requires stopping all other operations, which is highly impractical for an Internet webpage. Locking also can result in high tail latency, as all processes may need to wait on the slowest process to commit transaction and release its lock [9].

2.2 Possible Solutions

Some solutions to performance Some solutions to these problems still involve locks but provide performance benefits. For instance, instead of repeatedly modifying a value with spin-locking, one can wait until the cached value of the lock has changed, indicating that the lock is free, and then attempt to modify a value [3]. The problem of priority inversion can also be addressed with solutions like the priority ceiling protocol, which attempts to allow higher-priority tasks obtain locks first [1] or the *priority inheritance protocol*, in which some low-priority processes are temporarily given higher priority so they execute more quickly. In this latter protocol, if a low-priority process is currently holding onto a lock, but a higher-priority process is waiting on that lock, the low-priority process can "inherit" the high priority. The system will then prioritize this process, ensuring that it will release its lock quickly so that the higher-priority process can then have the lock [10]. However, these are imperfect solutions that still achieve less-than-ideal performance. The priority-ceiling protocol, for instance, significantly increases overhead and can incur unnecessarily high utilization, as processes must compare their relative priority [1].

Systems researchers look toward lock-free data structures and algorithms, which do not use critical sections [3], to avoid the aforementioned issues with locks. These data structures can guarantee progress and are immune to deadlock and tolerant of priority inversion [6]. In the literature, they are often used interchangeably with with “nonblocking” [6], as they guarantee that some non-faulty processes on them will complete in a finite amount of time. (Making lock-free data structures *wait-free*, i.e. guaranteeing that all processes will complete in a finite amount of time, often involves too much overhead to be practical [8].)

3 Lock-Free Logic and Examples

3.1 Compare-and-Swap

Lock-free data structures and algorithms often rely on commonly used primitives available on most processors, such as *Compare-and-Swap (CAS)*. CAS “takes three arguments: the address of a memory location, an expected value, and a new value.” It compares the value currently stored in the memory location to the expected value, and if they match, it swaps, or writes, the new value into the memory location. Otherwise, it fails, and the memory location is left unchanged [6]. This comparison allows processes to see whether other processes have modified shared memory. CAS is commonly used with *retry loops* [1], in which CAS is called repeatedly until it successfully returns, signifying that memory has been safely swapped.

3.2 Lock-Free Data Structures

Concurrent linked lists can successfully be implemented with CAS and other widely available synchronization primitives. Each concurrent access to the list uses a cursor, a group of three pointers that help track whether other processes have modified parts of the list close to its current node. With CAS, cursors can also help update auxiliary nodes in the list, and CAS is powerful enough to guarantee that this linked list implementation will be non-blocking. The general strategies behind this linked list implementation can be extended to create other lock-free data structures, such as dictionaries and binary search trees [8].

CAS and retry loops can also be used to construct lock-free, dynamically sized data structures, such as resizable lists. Processes should be able to concurrently insert nodes into, delete nodes from, and search through lists. The key logic of one CAS-based algorithm is to use CAS to determine whether a node has changed while it traverses a list. If CAS fails, indicating that the node has been changed by a concurrent process, it restarts its traversal process. CAS is also used to check whether a process has already deleted a node that another process is simultaneously attempting to delete, thus preventing repeated deletions and safely preparing the newly deleted node for reuse. For insertions, CAS is used in a retry loop that ensures that a list item will be correctly inserted. CAS also helps guarantee the linearizability of these lists, as the insert and delete operations have linearization points right after successful executions of CAS. These CAS-based lists can then be used to construct dynamic hash tables [5].

3.3 Lock-Free Memory Allocation

Beyond simple data structures, similar approaches can be used to implement algorithms, such as malloc and free. One possible implementation of the malloc function uses CAS in multiple places, and for brevity, these are two illustrative examples:

The malloc algorithm grants blocks of memory by allocating memory from superblocks, which contain pointers to the next available block of memory. Malloc reserves a block of memory with a retry loop that records the current pointer value and decrements the number of available credits from the superblock. It uses CAS to then check the new pointer value. If the pointer value is the same, malloc will successfully reserve the block and update the pointer accordingly. If the pointer value has changed, which means that another concurrent malloc has reserved that memory block, it will then retry reserving the block until there are no more available blocks of memory in that superblock.

Sometimes, if all current superblocks are full, i.e. the heap's superblock pointer is null, malloc may need to allocate a new superblock, so CAS is also used to ensure that multiple superblocks aren't created simultaneously. CAS checks to see that during the process of creating the new superblock, the pointer is still null. If the pointer now points to a superblock, this superblock must have been newly created or freed by a concurrent process, and should be used by malloc instead.

The free function can similarly be made lock-free [6]. Other simple primitives like Load-Linked and Store Conditional (LL/SC) can replace CAS in both lock-free malloc and free, which is highly desirable since the major processor architectures (such as Intel [2]) have one of these primitives [6].

4 Challenges of Lock-Free Strategies

4.1 Additional Overhead

The example of a lock-free linked list demonstrates that implementing even a relatively simple concurrent data structure requires much more overhead and programming difficulty than a non-concurrent one. Typically, deleting a node from a linked list is very simple, as one only needs to change the appropriate pointers to effectively remove the node. However, with a concurrent linked list, deletion is much more challenging, as other processes may currently be on the deleted node and require the node's pointer to continue traversing through the list.

Therefore, in order for a concurrent linked list to work, developers must maintain the contents of the deleted node while still changing the pointers. Additionally, deleting a node may affect the synchronization of other processes that are concurrently inserting or deleting adjacent nodes. Even more overhead is required to address this issue, as in between each node must be an auxiliary node, with next pointers, and an extra field must be added to normal nodes. To allocate and reclaim cells, an additional concurrent object, with new operations, must be created to track free cells [8].

While this challenge of increased overhead due to concurrency is not unique to lock-free strategies, it is important to acknowledge the difficulty of guaranteeing safety with these strategies. Lock-based approaches, while incurring higher latency, can often be comparatively simpler to implement. For instance, a lock-based linked list simply uses "hand-over-hand" fine-grained locking [4], in which a node and its adjacent nodes are locked during concurrent operations. Therefore, as the linked list example illustrates, there can be tangible tradeoffs to using a lock-free scheme over a lock-based one.

4.2 The ABA Problem

While CAS has the advantage of being widely supported [6], a major disadvantage is that if not used carefully, it may result in the *ABA problem*. The ABA problem occurs when a process reads a value (e.g. "A") but is then interrupted by another process. This new process then changes the value to a different value (e.g. "B"), but before the original process can resume, the value is changed back to "A." Therefore, when the original process checks the value again, it will see that the value is "A," and CAS will succeed. However, it should ideally fail instead, since the value was actually modified by other processes. Without extra mechanisms in place, there is no way for the process to know that the value was changed [2].

Developers must carefully consider how to best prevent the ABA problem, since there are multiple strategies with different tradeoffs [2]. One simple strategy for preventing the ABA problem is using an extra version "tag" field [5]. The tag is incremented each time an area of memory is modified, so if CAS sees that the tag has changed, it knows that the memory has been previously modified and that it should fail [6]. Therefore, even if a section of memory is modified but later changed back, this tag will be lasting evidence of its previous modification. This strategy can be used to protect a wide variety of lock-free applications, such as linked lists [8], memory allocators [6], and hash tables [5]. However, an issue with this approach is that it needs to update the tag and value simultaneously, which often requires more complex primitives, such as CAS2 (*compare-and-swap two co-located words*), or additional primitives, which are not available on all hardwares [2, 6, 8].

There are more sophisticated solutions to the ABA problem that are feasible without relying on complex primitives. For instance, designs with *descriptor objects*, which contains information on the state of shared memory and records of pending operations. These descriptor objects allow a lock-free program to just use CAS to solve the ABA problem and update both the object and memory together. Based on experimental results, this scheme can have similar execution speeds to a design using CAS2 [2].

5 New Architectures for Lock-Based Programming

Given these challenges with CAS and complex primitives, some research on lock-free schemes focuses on designing new processor architectures that can better support lock-free programming. *Transactional memory*, for instance, is a multiprocessor architecture that enables programmers to write *lock-free transactions*, or sequences of machine instructions that can effectively replace locks and critical sections. Transactional memory guarantees that transactions will be serial, as different processors will observe committed transactions occurring in the same order, and atomic, as either the entire transaction commits or aborts and commits nothing [3].

A traditional design of transactional memory adds new primitive instructions for accessing memory in three different ways: a shared read, an exclusive read (indicating a likely write), and an exclusive tentative write. Transactional memory also adds new primitives for changing transaction state: *commit* (which either commits the tentative write or aborts), *abort*, and *validate* (which sees whether the current transaction has aborted). *abort* helps avoid the deadlock and convoying issues that lock-based mechanisms typically experience by exiting from transactions that have timed out. *validate* allows programmers to check the consistency (i.e. serial execution of transactions) of transactions [3].

Two advantages of transactional memory are that it can reduce programmer workload and outperform locking schemes. First, instead of writing the code that manages concurrency among parallel programs, the programmer only needs to write transactions and provide their isolation level, as the transactional memory system will take care of the synchronization itself [7]. Second, fewer memory accesses and no explicit locks make transactional memory especially efficient. While the aforementioned transactional memory design has the limitation of requiring hardware modifications, newer designs for transactional memory systems can also rely on purely software techniques. These new techniques can even outperform the aforementioned hardware design in some benchmarks, such as workloads with high contention [7].

6 Experimental Comparisons of Lock-Based and Lock-Free

Early attempts at implementing lock-free data structures in the early 1990s have often experienced performance issues. Without deadlock, convoying, or priority inversions, traditional locking mechanisms experimentally outperformed the new lock-free techniques [3]. However, more recent implementations of lock-free data structures, some of which have already been mentioned in this review, are competitive with, or even better than, optimal traditional locking approaches. The improved performance of lock-free data structures, objects, and memory can be demonstrated both through formal analysis and experimentation.

Formal analysis proves that lock-free retry loops are bounded (i.e. will eventually execute) if tasks are properly scheduled [1]. Therefore, if the cost of a lock-free retry loop is less than the cost of the overhead required for lock-based schemes (which is usually very high), a lock-free scheme should outperform a lock-based one.

An experiment with a real-time desktop videoconferencing system confirmed this formal analysis. In this system, one workstation compressed and transmitted audio and video samples to another workstation, which would decompress and display them. Using a lock-free, shared queue scheme resulted in significantly lower processor utilization (94% vs 99.4%) than even optimized lock-based schemes. The lower utilization was also evidence that the lock-based strategy has higher overhead, as it must update task deadlines each time shared objects are accessed. Additionally, the lock-free scheme had an extremely low interference rate (i.e. ratio of CAS retry loops that never succeeded) of 0.08%, met all task deadlines, and did not drop a single data sample in the pipeline [1].

Simulations can also show that transactional memory, implemented with snoopy caches, is as efficient as well-known, optimized lock-based techniques. A process can commit or abort a transaction without coordinating with other processes or writing to main memory. Based on multiple benchmarks, this optimized implementation of transactional memory requires fewer memory accesses and exhibits higher throughput than lock-based mechanisms [3].

Lock-based methods often experience high latency partly due to lock contention, in which processes must wait on the same lock. However, experimental results demonstrate that even with reduced or nonexistent lock contention, lock-free methods still have lower latency than lock-based ones. For example, an algorithm for a lock-free hash table experienced lower latency in both high-contention and low-contention scenarios [5]. The lock-free hash table achieved significantly faster execution times on varying distributions of insert, search, and delete operations than even a scheme using reader-writer locks. Even as the load factor increased, the lock-free method's CPU time stayed relatively similar [5]. Lock-free memory allocators achieved lower contention-free latency and higher space efficiency than even optimal lock-based schemes, making it likely to outperform even schemes using "the fastest lightweight lock to protect malloc and free" [6]. The lock-free memory allocators also achieved high scalability, as an increase in processes leads to a linear increase in speedup (some lock-based mechanisms, on the other hand, exhibit very little speedup and therefore poor scalability) [6].

Therefore, lock-free designs for a wide variety of purposes and applications can experimentally outperform their lock-based counterparts, and therefore are viable solutions to the performance issues that lock-based strategies often experience.

7 Practical Applications

Given their performance benefits, lock-free data structures are now used in a variety of practical computing applications, including web applications and internal memory allocators.

Videoconferencing systems must handle large quantities of real-time audio and video samples, placing them in different queues to be transmitted, digitized, and compressed, and these queues can be successfully made lock-free, capable of processing samples without losing a single one [1].

Memory allocation operations, such as malloc and free, have traditionally used locks to provide consistency of shared memory and allow for safe multithreading. However, it is possible to implement these operations without locks, thereby lowering latency [6].

Web applications with mirror sites, or multiple copies of a site that provide availability and load balancing, must ensure that changes to one mirror are eventually reflected in the other mirrors. These mirror sites are often used for forums, in which many users perform writes concurrently. For mirror sites, traditional two-phase locking is an overly strict form of concurrency control that results in high latency, as well as unnecessary blocking of other operations and sites. Lock-free approaches, however, take advantage of the fact that these mirror states do not need complete serializability to achieve reasonably low latency. They allow for slightly weaker concurrency, using a timestamping strategy that tracks and preserves causal relationships. Each site executes operations in receives immediately, but the site keeps record of version history, so it can properly recover old states and follow causal relationships. Eventually, the mirror sites will converge, and this lock-free strategy enables load balancing and short response times [9]. Ultimately, this is an example of how lock-free applications are not merely improvements upon lock-based strategies already in practice – they can be used in some scenarios where lock-based protocols are insufficient and infeasible.

8 Conclusion

Overall, lock-free data structures and algorithms offer safe (and oftentimes, more efficient) alternatives to lock-based mechanisms. These lock-free approaches typically rely on the compare-and-swap primitive, which is widely available on processor architectures. While programmers must be aware of specific challenges, such as the ABA problem, lock-free techniques are capable of being used in real-world practical scenarios. Additionally, while many lock-free techniques can provide the same guarantees as their lock-based counterparts, others may actually provide weaker guarantees of consistency and serializability, which could be exploited in scenarios that do not require perfect concurrency and replication.

Given their strong performance in building data structures, communications networks, and underlying systems logic, lock-free strategies represent promising advancements in a wide variety of concurrent applications. Lock-free schemes should be studied to not only develop performance improvements to lock-based ones, but also stretch the limits of current software and hardware and help us reimagine how to better implement processor architectures and distributed computing alike.

Acknowledgments

I would like to acknowledge Xiao Shi and Professor Y. Richard Yang for their guidance throughout this course and this literature review.

The formatting of this review is from a template by the USENIX website.

References

[1] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, may 1997.

[2] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, 2010.

[3] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, may 1993.

[4] Maurice Herlihy and Nir Shavit. chapter Linked Lists: The Role of Locking. Elsevier, 2008.

[5] 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*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery.

[6] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 35–46, New York, NY, USA, 2004. Association for Computing Machinery.

[7] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, Sep. 2008.

[8] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 214–222, New York, NY, USA, 1995. Association for Computing Machinery.

[9] Jiangming Yang, Haixun Wang, Ning Gu, Yiming Liu, Chunsong Wang, and Qiwei Zhang. Lock-free consistency control for web 2.0 applications. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, page 725–734, New York, NY, USA, 2008. Association for Computing Machinery.

[10] Xingyuan Zhang, Christian Urban, and Wu Chunhan. Priority inheritance protocol proved correct. *Journal of Automated Reasoning*, 64(1):73–95, 01 2020. Copyright - Journal of Automated Reasoning is a copyright of Springer, (2019). All Rights Reserved. This work is published under https://creativecommons.org/licenses/by/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2020-01-24.