General Techniques for Efficient Concurrent Data Structures

Yuanhao Wei

CMU-CS-23-125

August 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation (NSF) under grant numbers CCF-1901381, CCF-1910030, CCF-1919223, and CCF-2119069, and the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Postgraduate Scholarship – Doctoral(PGS-D) program. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, any government or any other entity.

Keywords: Concurrency, Lock-freedom, Snapshot, Memory Reclamation, Idempotence, Multiversioning

To my family.

Abstract

Scalable concurrent data structures are essential for unlocking the potential of modern multicore machines. This thesis presents techniques for enhancing existing concurrent data structures with several useful properties: lock-freedom, the ability to take consistent snapshots, and safe memory management. The goal is to make these techniques widely applicable, easy-to-use, theoretically efficient (i.e. fast in worst-case executions), and also fast in practice.

For lock-freedom, we present a new approach to lock-free locks based on helping, which allows the user to write code using the familiar interface of locks, but run it in a lock-free manner. This thesis presents some key techniques that make lock-free locks practical and more general. We show that our lock-free locks can significantly outperform traditional blocking locks in certain workloads.

We also present an approach for efficiently capturing a consistent view of a concurrent data structure at a single point in time. This is useful for computing linearizable multi-point queries such as searching for a range of keys, finding the first key that matches some criteria, or checking if a collection of keys are all present. Importantly, our approach preserves the time bound and parallelism of the original data structure. It can be applied to both lock-based and lock-free data structures and is compatible with the lock-free locks approach introduced in the first part of the thesis.

Finally, we present a safe automatic memory reclamation approach for concurrent programs, and show that it is both theoretically and practically efficient. Our approach combines ideas from reference counting and hazard pointers in a novel way to implement concurrent reference counting with wait-free, constant-time overhead. It overcomes the limitations of previous approaches by significantly reducing modifications to, and hence contention on, the reference counts. We further generalize this approach to allow a variety of safe memory reclamation (SMR) schemes to be used as a substitute for hazard pointers. This augments the SMR schemes with ease-of-use while maintaining their performance profiles in terms of time and space.

Acknowledgement

I am extremely grateful to my advisor, Guy Blelloch, for all the guidance and encouragement throughout my graduate studies. I am very fortunate to have had the chance to work closely with Guy and learn from his breadth of technical expertise. Working through problems together on the white board made up some of my fondest memories at CMU. Thank you Guy for all the time you spent with me, for all the insightful advice and for being an amazing role model.

I would also like to thank the other members of my thesis committee, Philip Gibbons, Andy Pavlo, Faith Ellen, and Panagiota Fatourou for all the detailed and valuable feedback on this thesis.

I am especially indebted to Faith who introduced me to research in concurrent algorithms back when I was an undergraduate student at the University of Toronto. Faith spent many hours patiently editing my early writings, listening to my half baked ideas, and guiding me through the research process.

Throughout my PhD, I have had the chance to collaborate with many amazing researchers who I learned a lot from and enjoyed thinking through problems with. Thank you to Peter Chen, Naama Ben-David, Guy Blelloch, Yihan Sun, Michal Friedman, Ian Mertz, Toniann Pitassi, Erez Petrank, Panagiota Fatourou, Eric Ruppert, Daniel Anderson, Laxman Dhulipala, and Magdalen Dobson for all the insightful discussions and fruitful collaborations. I would additionally like to thank Naama for her boundless optimism, especially before deadlines, and Daniel for teaching me how to write modern and efficient C++.

I would also like to thank my friends and fellow CMU students Paul Gölz, Laxman Dhulipala, Daniel Anderson, Naama Ben-David, Pratik Fegade, Jin Kyu Kim, Logan Brooks, John Shi, Alvin Shek, Danniel Yang, Zhenyu Yang, Eric Zheng, Sam Westrick, Davis Zhang, Anup Agarwal, Goran Zuzic, Freddie Feng, Yong Kiam Tan and many more for all the memorable moments and for making this journey more enjoyable. To Laxman, thank for you being an amazing roommate and a constant source of inspiration. I had a lot of fun solving puzzles together and discussing all sorts of topics. To Paul, thank you for being there whenever I needed someone to talk to and for the wonderful dinner parties. Thank you also to Pratik for the fun office debates, Alvin, Zhenyu and Anup for waking up early to play badminton, Yong for teaching me about classical music, Danniel for being a longtime friend, and Davis for the surprise birthday party.

Finally, I would like to thank my parents, my little brother, and my girlfriend, Bessie Xue, for their unconditional support and encouraging me to pursue my dreams. I could not have come this far without you.

Contents

Chapter 1

Introduction

Given the widespread use of multiprocessor machines, there has been significant work, especially in recent years, on designing efficient concurrent data structures. Concurrent data structures are ones that allow multiple processes to access and make changes to them simultaneously. They are a crucial part of many applications such as database systems, operating systems, parallel runtime environments, and memory allocators. Nowadays these data structures need to scale well past a hundred cores to fully utilize the potential of commodity multiprocessor machines. However, designing concurrent data structures has proven to be notoriously difficult due to the complex interleaving of instructions from processors executing asynchronously. It becomes especially difficult if we want to support more advanced features such as lock-freedom, consistent snapshots, and safe memory reclamation. As evidence for the difficulty, the first lock-free implementation of a binary search tree was not until 2010 [62], building on decades of work on concurrency and lock-freedom. Prior to the work in this thesis, these more advanced features could only be added efficiently by experts in the field. Given how difficult a single concurrent data structure is to develop, redesigning each of them to support these extra features is not scalable.

Rather than working on specific data structures, this thesis develops several general techniques for enhancing existing data structures with these more advanced features. *This thesis contends that with the right abstractions and algorithms, general techniques can be very efficient, often outperforming hand-designed data structures.* The algorithms we develop can be intricate and subtle, but we abstract this complexity away from the user by designing a clean interface for each of our techniques. Having these easy-to-use interfaces makes concurrent programming more accessible to non-experts. It also makes reasoning about the algorithm and proving correctness easier and more modular. *The goal of this thesis is to simplify the design of new concurrent data structures by developing general techniques and libraries usable by experts and non-experts alike.*

We validate the practical performance and scalability of these techniques by running them on a multicore machine with over a hundred cores. To ensure robust performance across thread schedules and inputs, we also prove theoretical bounds on time and space for data structures written using our general techniques.

Lock-freedom. Mutual exclusion and locks are perhaps the most well-known and widely used tools for concurrent programming. However, traditional implementations of locks require threads to wait if the requested lock is already taken. This is undesirable in asynchronous systems where the thread holding the lock could be slow, paused by the system scheduler, or even crashed. Lock-free programming was introduced to remedy these issues. *Lock-freedom* is a technical term, and intuitively, it means some thread will make progress regardless of how they are scheduled.

While many programmers are familiar with mutual exclusion and locks, techniques for lock-free programming are less well-known and more complex. We developed the first practical "lock-free" (in the formal sense) implementation of locks which allows the user to write code using fine-grained locks, but then run it in a lock-free manner. This approach is purely library based and does not require the user to know anything about lock-free programming. Furthermore, lock-free data structures written using this technique can be highly efficient. We applied the technique to a variety of lock-based data structures for trees, lists, and hash tables, and found that our lock-free versions perform up to 2.4x faster the original lock-based versions in oversubscribed environments. Our lock-free versions were also competitive with state-of-the-art lock-free data structures for trees, lists and hash tables.

When a process wishes to take a lock, the idea is that instead of waiting, it tries to help the process that currently has the lock exit its critical section. The challenge is in performing the helping idempotently. This means each critical section should appear to have executed only once even though it might have been helped by multiple processes. We present a practically efficient solution to this problem that involves using a shared log to ensure all processes agree on the outcome of each read and write.

Consistent snapshots. Atomic snapshots capture the state of a concurrent data structure at one instance in time. This can be useful for debugging as well as for implementing linearizable multi-point queries which require looking at multiple keys in the data structure. Examples of such queries include searching for a range of keys (range query), finding the first key that matches some criteria, or checking if a collection of keys are all present. Snapshots can be difficult to implement because the state of the data structure can be changing while the snapshot is being taken. For this reason, many data structures only support non-linearizable multi-point queries (e.g. a range query that may return a set of nodes that were never simultaneously in the data structure).

This work presents a general technique for adding linearizable snapshots and multi-point queries to existing concurrent data structures. This transformation maintains the time bounds and progress properties (e.g. lock-freedom/wait-freedom) of the original data structure, and it also ensures that multi-point queries are wait-free. Moreover, the technique provides good time bounds, with multi-point queries taking time proportional to their sequential complexity plus a contention term representing the number of update operations concurrent with the query. This technique applies to a wide range of data structures, both lock-based and lock-free, and can be used to support arbitrary multi-point queries. Despite its generality, it is also extremely efficient; in our experimental evaluation, it often outperforms data structures specifically designed to support fast range queries. Our technique leverages an idea called multiversioning, which involves keeping around historical versions of each object.

Safe memory reclamation. One of the main challenges holding back the widespread adoption of concurrent data structures in practice is safe memory management. When a memory location is removed from a data structure, it is not safe to immediately reuse it for another purpose because a concurrent operation could be working on an outdated view of the data structure and might access that memory location in the future. In the research community, concurrent data structures are often designed without any memory reclamation and this step is treated as an orthogonal problem. This causes a dilemma for practitioners who need to apply a memory reclamation scheme themselves in order to deploy the data structure in their system. Several manual memory reclamation schemes have been proposed, but these are difficult to apply correctly, even for experts. Automatic approaches based on concurrent reference counting are much easier to use, but they have traditionally been seen as being slow and not scalable.

In this work, we present the first automatic memory reclamation scheme (based on reference counting) that adds only constant time overhead while using only instructions available on modern machines. In practice, it is as fast as Hazard Pointers (a widely used manual memory reclamation scheme), while being significantly easier to use and less error prone. It also outperforms all existing concurrent reference counting implementations. Our technique involves using a novel generalization of hazard pointers to defer reference-count decrements until no other process can be incrementing them. For efficiency, we also defer or elide reference-count increments for short-lived references. We further generalize this approach to allow a variety of safe memory reclamation (SMR) schemes to be used as a substitute for hazard pointers. This augments the SMR schemes with ease-of-use while maintaining their performance profiles in terms of time and space.

1.1 Lock-freedom

Lock-free algorithms, or data structures, are guaranteed to make progress even if processes crash or are delayed indefinitely. They are, however, burdened with some issues. One important issue is that they tend to be significantly more complicated than their lock-based counterparts. Even basic data structures such as stacks, queues, and singly linked lists can lead to non-trivial lock-free algorithms with subtle correctness proofs. More sophisticated data structures, such as binary trees and doubly linked lists, become considerably more complicated. If one needs to atomically move data between structures, lock-free algorithms become particularly tricky. Developing efficient algorithms with fine-grained locks is not necessarily easy, but is typically much simpler.

Another issue is performance. The relative performance of lock-free vs. lock-based algorithms depends on the environment in which they are run. Several papers demonstrate that lock-based concurrent algorithms can be faster [8, 48, 83, 162]. However, the experiments described in these papers are typically run in rarified environments in which all processes are dedicated to the task, often pinned to dedicated cores. They are also set up to have no page faults or other significant delays. In such environments, it is not surprising that algorithms using fine-grained locks do well. Some have noted, however, that in environments with oversubscription (more processes than cores) lock-based algorithms can suffer due to threads getting descheduled while holding a lock [48]. In the Linux operating system, for example, there has been a push to use lock-free structures (at least for reads) due to mixed workloads and the unpredictability of demand [111]. Also, of course, lock-based algorithms can become blocked in environments where processes can be faulty.

In summary, for robustness in multiprogramming environments, or for peace of mind in general, lock-free algorithms can have a significant advantage, but they come at the cost of more subtle and complicated designs, especially when used for more advanced data structures. Due to the tradeoffs, there is no universal agreement on whether lock-based or lock-free algorithms are better—some algorithms are lock-free [37, 62, 80, 81, 122, 162] and others use fine-grained locks [16, 33, 58, 83, 99, 104, 110, 139]. A third choice is to use transactional memory, but this has not yet shown itself to be competitive with either lock-free or lock-based approaches.

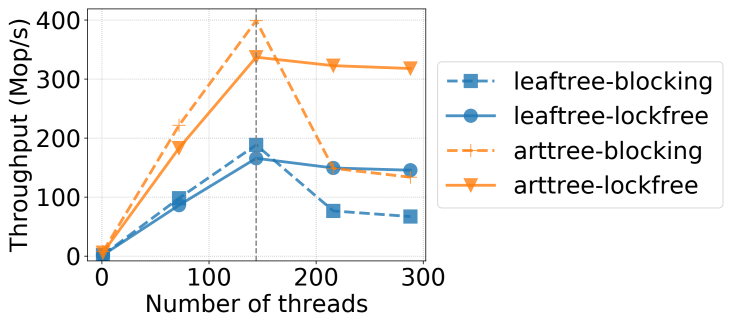


Figure 1.1: Comparing traditional blocking locks with lock-free locks on a concurrent binary search tree (`leaftree`) and a concurrent radix tree (`arttree`). Workload consists of 50% updates and 50% lookups on a tree initialized with 100K keys.

In Chapter 3, we describe and study an approach that gets the best of both worlds—i.e., allow one to program with fine-grained locks while getting efficient lock-free behavior. It is based on the idea of having processes help complete each other's critical sections rather than passively waiting for the lock. This means that a critical section must be *idempotent* so that even if it is performed once by the original process and many times by helping processes, it still appears to only have executed once. A crucial part of our approach is a general mechanism for making critical sections idempotent without adding much overhead. The idea is to maintain a shared log among processes helping to run the same code. Processes use this log to agree on the results of shared memory operations, as well as other events such as memory allocation. To achieve agreement, we use the compare-and-swap instruction – whichever process commits first wins, and all others take the committed value instead of their attempted commit. For example, each read operation attempts to commit the value it read. This way, all processes read the same values and follow the same control flow throughout the code even though they are running in an arbitrary interleaved manner.

In our approach, the user can write standard code based on fine-grained locks, and using a simple library interface, get lock-free behavior. We have implemented our approach as a C++ library called Flock. Based on the library we have implemented several data structures based on try-locks, including singly linked lists, doubly linked lists, binary trees, balanced blocked binary trees, (a,b)-trees, hash tables, and adaptive radix trees (ART). We compare performance of our versions in lock-free mode and blocking mode to the most efficient existing data structures we found, both lock-based and lock-free. The lock-based data structures generally perform slightly better under controlled environment with one process per processor, but perform significantly worse with oversubscription (multiple processes per processor). Comparing running our algorithms in lock-free vs. blocking mode, the lock-free performance rarely has more than 10% overhead, and typically much less. However, with oversubscription, the lock-free mode greatly outperforms the blocking mode by up to 2.4x. Figure 1.1 shows an example of this by comparing lock-free locks with traditional spin locks on a concurrent radix tree [104] and a leaf-oriented binary search tree, both of which use fine-gained locking.

Contributions. Our contributions include the following:

- A practical approach to achieving idempotence in general code
- A new algorithm for lock-free locks
- A general library-based interface to support our ideas, which we used to implement the first practical lock-free adaptive radix tree.

1.2 Consistent Snapshots

Many applications that use concurrent data structures require querying large portions of the data structure. For example, one may want to filter all elements by a certain property, perform range queries, or simultaneously query multiple locations. However, such multi-point queries have been notoriously hard to implement efficiently. Although it is easy to support multi-point queries by locking large parts of the data structure, this approach lacks parallelism. Some concurrent data structures resort to multi-point queries that provide no guarantee of atomicity [129, 130]. Other efforts have implemented specific multi-point queries (e.g., range queries, iterators) [3, 7, 36, 42, 66, 67, 134].

A general way to support efficient multi-point queries is to provide the ability to take a *snapshot* of the data structure. Conceptually, a snapshot saves a read-only version of the state of the data structure at a single point in time [2, 6, 70]. Multi-point queries can be performed by taking a snapshot and reading the necessary parts of that version to answer the query, while updates run concurrently. Snapshots are also used in database systems for multiversioning and recovery [25, 53, 124, 131, 136, 144, 171], and in persistent sequential data structures [59, 60, 147]. However, known approaches for taking snapshots either limit the programming model (e.g. to be purely functional [19, 56]), use locks with no progress guarantees [25, 98, 124], or are lock- or wait-free but have large running times [2, 32, 65, 69, 92].

Given a concurrent data structure, Chapter 4 presents an approach for efficiently taking snapshots of its constituent Compare&Swap (CAS) objects. More specifically, it supports a constant-time operation that returns a snapshot handle, which represents the point in time the snapshot was taken. This snapshot handle can later be used to read the value of any base object at that time. Reading an earlier version of a base object is wait-free and takes time proportional to the number of successful writes to the object since the snapshot was taken. Importantly, this approach preserves all the time bounds and parallelism of operations supported by the original data structure. For example, Table 1.1 shows the time bounds achieved by using this technique to add linearizable query operations to several popular lock-free data structures.

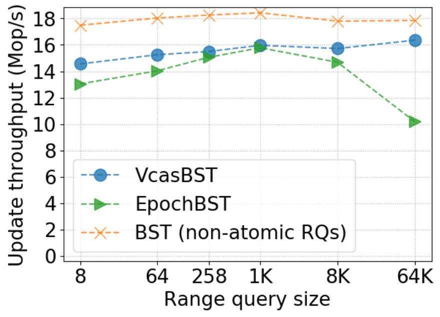
Original Data Structure	Operation	Our Time Bounds	Parameters
Michael Scott Queue [118]	`select`(i): `enqueue/dequeue`:	$O(i+c)$ same as original	c: number of dequeues concurrent with the query
Harris Linked List [80]	`range`(s,e): `multisearch`(L): `ith`(i): `insert/delete/lookup`:	$O(m+P+c)$ $O(m+P+c)$ $O(i+P+c)$ same as original	m: number of keys in list c: number of inserts/deletes concurrent with the query P: number of processes
Ellen, Fatourou, Ruppert, Breugel BST [62]	`successor`(k) `multisearch`(L): `range`(s,e):	$O(h+c)$ $O(\lvert L\rvert \times h+c)$ $O(h+K(s,e)+c)$	m: number of keys in BST h: height of tree. $K(s,e)$: number of keys in BST between s and e
Brown, Ellen, Ruppert Chromatic Tree [38]	`height()`: `insert/delete/lookup`:	$O(m+c)$ same as original	c: number of inserts, deletes, rotations concurrent with the query P: number of processes

Table 1.1: Time bounds for various operations on concurrent queues, lists, and BSTs using our snapshot approach. All the operations we give bounds for are not supported by the original data structure. All parameters other than the contention term c are measured at the linearization point of the operation. The contention term c measures interval contention meaning that it counts the number of update operations that overlap with the query operation. In the case of Chromatic Tree, the height of the tree $h \in O(\log(m)+P)$.

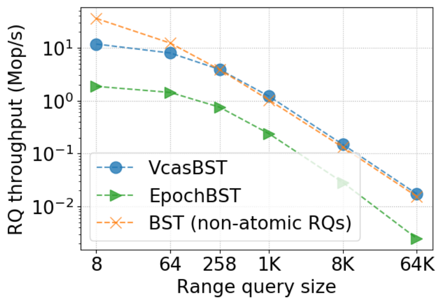
The idea is to keep around historical versions of each CAS object in a version list, where each version is tagged with the timestamp at which it was written. Taking a snapshot simply involves reading and incrementing the global timestamp, and then reading the desired version lists using this timestamp. The algorithmic challenge is in assigning up-to-date timestamps to newly added versions in a lock-free manner. This is tricky to do in the concurrent setting because the timestamp assignment needs to be done atomically with the insertion into the version list. We present a new, constant time algorithm for this which first inserts a version with a temporary timestamp and uses helping to update it to the final timestamp. Helping is done carefully to make sure that no processes uses the version list while it is in an intermediate state. This algorithm allows us to achieve the bounds in Table 1.1, which have not previously been achieved for any of the listed data structures.

In Chapter 5, we further extend this approach to work for lock-based data structures, as well as those written with the lock-free locks introduced in Chapter 3. We also present several important practical optimizations for avoiding indirection and reducing contention when accessing the global timestamp.

To evaluate the performance of our snapshotting approach, we apply it to a wide range of lock-based and lock-free data structures. Experiments show that the overhead of supporting linearizable snapshots is low across a variety of workloads. Moreover, range queries on the trees built from our snapshots perform as well as or better than state-of-the-art concurrent data structures designed to support atomic range queries. For example, Figure 1.2 presents performance numbers for a snapshottable version (`VcasBST`) of a lock-free BST (`BST`) designed



(a) Update Throughput



(b) Range Query Throughput

Figure 1.2: Evaluating the performance of our snapshotting approach on a workload with 36 threads performing updates (half insert, half deletes) and 36 threads performing range queries on a tree initialized with 100K keys.

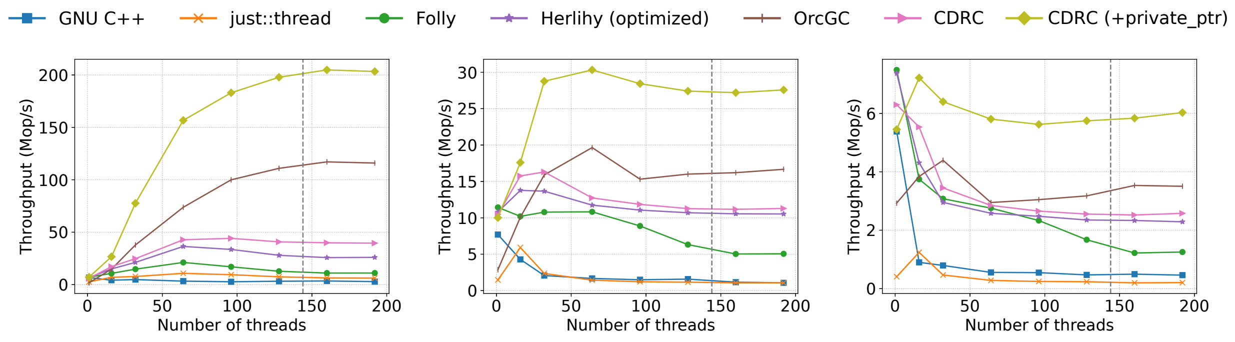
using Brown, Ellen and Ruppert's tree update template [38]. Range queries on the original BST are not linearizable, but we include it in Figure 1.2 as an upper bound on how fast a linearizable version can be. Our VcasBST comes close to this upper bound in terms of both update and range query throughput. It is also consistently faster than a state-of-the-art BST supporting linearizable range queries called EpochBST [7]. More details on this experiment, as well as a more complete experimental evaluation, can be found in Section 4.9.

Contributions. In summary, our contributions are:

- A simple, constant-time approach to take a snapshot of a collection of CAS objects.
- A extension of this approach to support lock-based data structures (works with both lock-free and blocking locks).
- A technique to use snapshots to implement linearizable multi-point queries on many lock-based and lock-free data structures.
- An easy-to-use, portable library, VERLIB, for adding linearizable snapshots to existing or new concurrent data structures.
- Experiments showing our technique has low overhead, often outperforming other state-of-the-art approaches, despite being more general.

1.3 Safe Memory Reclamation

Memory reclamation, the problem of freeing allocated memory in a safe manner, is essential in any program that uses dynamic memory allocation. A block of memory is safe to reclaim only when it cannot be subsequently accessed by any thread of the program. Determining exactly when this is the case is, however, a difficult problem for mutlithreaded programs which could be sharing, copying, or modifying references to the same memory blocks concurrently. One solution is to rely on a garbage collector [93], though it is not always possible, most-efficient, or most-flexible. In languages without built-in garbage collectors, memory reclamation for concurrent programs, often called *safe memory reclamation* (SMR), is a non-trivial and extensively studied problem. A crucial difficulty in the concurrent setting is the possibility of read-reclaim races [82].



(a) 1% pushes/pops (b) 10% pushes/pops (c) 50% pushes/pops

Figure 1.3: Benchmark results comparing various concurrent reference counting implementations when applied to a concurrent stack.

Such a race is between a process that reads and follows a pointer to an object and another that reclaims and reuses the corresponding memory.

Safe memory reclamation techniques can be broadly divided into two categories, manual and automatic. With manual techniques, the user is responsible for freeing objects. To protect against read-reclaim races, this is often performed with a *retire* operation, which defers the reclamation until it is safe, i.e., until no other thread is reading that object. Manual techniques are often fast but difficult to use, leading to subtle and hard to reproduce bugs even in code written by experts. Automatic techniques are similar to what can be found in garbage collectors, but without the ability to scan processor private root sets (registers, stacks, etc.). A common automatic technique is reference counting [51, 87, 102, 115, 135, 154, 160], which requires very few modifications for programmers to integrate into their code, and provides memory safety and leak freedom automatically as long as the programmer either does not create reference cycles or breaks such cycles before they become unreachable. C++ and Rust have had reference counting in the form of smart pointers for over a decade. Owing to their ease of use, there has been an increase in interest in thread-safe, atomic reference-counted pointers, as evidenced by their inclusion in the most recent C++ standard (C++20). However, reference counting has traditionally been considered to be inefficient in the concurrent setting due to frequent increments and decrements of shared counters [82]. Indeed, we found that the current implementation in the widely used GNU C++ library does not scale.

Chapter 6 proposes an efficient approach to automatic memory reclamation based on a novel combination of reference counting and manual SMR. It makes several advances to make library-based concurrent reference counting both theoretically efficient and more practical, while preserving its ease-of-use. Theoretically, it shows the first solution with constant expected time overhead using only single word atomic primitives and only delaying $O(P^2)$ decrements at each moment in time (meaning the reference counts might be $O(P^2)$ more than the actual number of references).

We have implemented our technique as a C++ library called CDRC [1] and show that it is more efficient than existing optimized libraries for atomic reference-counted pointers [46, 64, 169].

[1]Available at https://github.com/cmuparlay/concurrent_deferred_rc

For example, Figure 1.3 compares CDRC with widely used commercial and open source libraries and shows that it is significantly faster under both read-dominated and update-heavy workloads. A more detailed experimental evaluation can be found in Section 6.7

In Chapter 7, we generalize the CDRC technique so that it is capable of turning a wide variety of manual SMR techniques into automatic techniques based on reference counting. We show experimentally that our automatic techniques have similar throughput and memory usage to their manual counterparts, while being safer to use and applicable to more data structures.

Contributions.

- A theoretically and practically efficient algorithm for concurrent reference counting.
- A generalization of the previous algorithm to convert a wide range of manual SMR techniques into automatic SMR techniques using reference counting.
- Experiments showing that our automatic techniques have similar throughput and memory usage to their manual counterparts. (This represents a 2x-3x throughput improvement over existing concurrent reference counting implementations.)

1.4 Outline and Thesis Statement

Outline. Part I describes our completed work on lock-free locks. Part II describes our completed work on linearizable snapshots. Part III describes our completed work on safe automatic memory reclamation based on reference counting. We conclude in Part IV.

The results in this thesis are primarily based on previous publications, and also include new results that are unpublished. Many other published papers [18, 19, 20, 24, 27, 28, 73, 114, 165, 166] by the thesis author are not included in this thesis. The papers included in this thesis are listed below.

- [21] Lock-Free Locks Revisited. Naama Ben-David, Guy Blelloch and Yuanhao Wei. In *Principles and Practice of Parallel Programming (PPoPP)*, 2022. (received a best paper award, full version in [23]) Included in this thesis in Chapter 3.
- [164] Constant-time snapshots with applications to concurrent data structures. Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert and Yihan Sun. In *Principles and Practice of Parallel Programming (PPoPP)*, 2021. (full version in [163]) Included in this thesis in Chapter 4.
- [4] Concurrent deferred reference counting with constant-time overhead. Daniel Anderson, Guy E. Blelloch and Yuanhao Wei. In *Programming Language Design and Implementation (PLDI)*, 2021. Included in this thesis in Chapter 6.
- [5] Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting. Daniel Anderson, Guy E. Blelloch and Yuanhao Wei. In *Programming Language Design and Implementation (PLDI)*, 2022. Included in this thesis in Chapter 7.

This thesis provides evidence to support the following statement:

Thesis Statement. *General techniques, along with appropriate abstractions and library implementations, can greatly simplify the design and implementation of efficient concurrent algorithms.*

Examples covered in this thesis are techniques for lock-freedom, obtaining a consistent view of a data structure, and automatic memory reclamation.

Chapter 2

Preliminaries

We consider an asynchronous shared memory system with n processes. Processes communicate by accessing shared memory via the following atomic primitives: *read, write*, *compare-and-swap (CAS)*, *fetch-and-add (FAA)*, and *exchange*[1]. The compare-and-swap primitive takes three parameters, a memory location, an expected value and a new value. It writes the new value in the memory location and returns true if the current value of the memory location matches the expected value. Otherwise, it returns false and leaves the memory location unchanged. The fetch-and-add primitive takes two parameters, a memory location and an integer a. It atomically reads the current value v stored in the memory location, updates the memory location to store $v + a$, and returns v. The exchange operation takes as input a memory location and a value v. It writes v into the memory location and returns the value that was overwritten. The read, write and compare-and-swap (CAS) primitives are sufficient for all the algorithms in Parts I and II. Part III also makes use of the exchange and fetch-and-add primitives. All these primitives are supported in hardware by modern processors. Chapter 4 considers a special type of memory location that only support read and compare-and-swap operations, referring to them as *CAS objects*.

An *execution* is an alternating sequence of configurations and *steps*. Each configuration provides a global view of the system at some point in time and each step specifies a primitive (on either local or shared memory), its arguments, its return values, and the executing process. The steps taken by a process in an execution implement *operations*. An *event* is the *invocation* or *response* of an operation, which specify its arguments and return values, respectively, as well as its calling process. The first step of an operation in an execution is associated with its invocation and its last step is associated with its response. The *execution interval* of an operation starts from its invocation and ends at its response. A *history* is a sequence of events, and can be derived from an execution E by including the invocations and responses of operations in the order their associated steps appear in E. An execution is *valid* if it is consistent with the semantics of the memory operations.

A data structure is a set of operations. Each operation is specified by a *sequential specification*, which defines its expected behavior in an execution in which the executing process's steps are not

[1]Also known as *swap* and *fetch-and-store (FAS)*

interleaved with the steps of any other process. An implementation of a data structure specifies code for processes to run for each of its operations. The correctness condition we consider is called *linearizability* [88]. An execution α is *linearizable* if, for every complete operation *op* in α (as well as for some of the uncompleted operations), we can assign it a *linearization point* within its execution interval, so that in the sequential execution defined by the linearization points, each operation has the same response as in α. Intuitively, this means that each operation appears to take effect atomically at some point during its execution interval. An implementation is *linearizable* if all its executions are linearizable.

We do not assume any bounds on the relative speeds of the processes, which means they can be arbitrarily slow or even crash (i.e. not take any more steps in the execution). An implementation of a data structure D is *lock-free* if, in any infinite execution in which processes follow this implementation, infinitely many operations complete. Intuitively, it means whenever there are ongoing operations, one of them will complete after a finite number of steps regardless of how processes are scheduled. *Wait-freedom* is a stronger property which requires all operations to complete within a finite number of their own steps. The *time complexity* of an operation is the number of steps (both local and shared) performed by that operation before it completes in a worst-case execution. We count both local and shared memory towards *space* usage.

We say a memory location suffers from the *ABA problem* in some implementation if it is possible for the value stored in that memory location to go back to what it was at some previous point in some execution of this implementation. We say an implementation suffers from the ABA problem if there is some memory location that suffers from the ABA problem in that implementation. An implementation is *ABA-free* if it does not suffer from the ABA problem.

In our experiments, concurrent operations are executed by user-level threads, so we often use the term *thread* instead of *process* in the experimental sections. Appropriate memory barriers and fences were added to all our code to prevent instructions from being reordered.

Part I

Lock-freedom

Introduction

Lock-freedom is a useful property that ensures system-wide progress regardless of how processes are scheduled. However, lock-free algorithms can be significantly harder to design than lock-based ones. In this part, we simply this process by presenting a practical "lock-free" implementations of locks which allows the user to write code using fine-grained locks, but then run it in a lock-free manner.

Chapter 3

Lock-free Locks

3.1 Introduction

In this chapter, we describe and study a practical implementation of lock-free locks which allows one to program with fine-grained locks while getting efficient lock-free behavior. The idea of lock-free locks were first proposed by Turek, Shasha and Prakash [159] and independently Barnes [14] (henceforth called the TSP-B approach). The high-level idea behind the TSP-B approach is that when a thread takes a lock it leaves behind a descriptor that allows other threads that want the lock to help it complete its protected code and free its lock. Our implementation is based on this idea, however, we extend it significantly with several important new ideas to make it practical and more general.

The general idea of using descriptors for helping is now widely used in the implementation of specific lock-free applications, such as multiword-CAS [68, 78, 81, 161], other multiword operations [37], software transactional memory [72, 143, 149], and specific data structures [20, 54, 62, 148, 170]. Despite the use of descriptors for helping in specific applications, prior to this work, we know of no general implementations of lock-free locks. Most of the papers cited above mention the TSP-B approach, but describe it as impractical and often use it as motivation for their more specific approach. The issue is that the TSP-B approach requires translating code in the lock into a form such that every read or write effectively requires saving the context of the process (program counter and local variables) so that others can help it run from that point. Such code can be very inefficient even when no helping occurs. Equally importantly, it makes the approach very difficult and clumsy to use without a special-purpose compiler. Their approach also constrains the code inside the locks to only allow race-free reads and writes to shared memory.

The key contribution of this chapter is an approach to avoid the "context-saving" on each memory operation, making the approach practical, and additionally making it more general. In our approach, the user can write standard code based on fine-grained locks, and using a library interface, get lock-free behavior as long as the code is deadlock-free. Our library preserves the correctness of the lock-based data structure because all the helping is done idempotently, so each critical section appears to have only executed once. Beyond being efficient and offering a simple library-based interface, our approach generalizes the TSP-B approach by (1) allowing races in the locked code, (2) supporting memory allocation and freeing in the locked code, and (3) supporting try locks, which we demonstrate are much more efficient than the standard strict locks. The advantage of a try lock is that it returns false if the lock is currently taken, giving the user the flexibility of either trying again or performing a different operation.

Our approach is based on a new technique to achieve idempotence. Intuitively, idempotent code is code that can be run multiple times but appears to have run once [17, 34, 49, 50]. Such code is important in the TSP-B approach, since multiple helpers could run the same locked code when helping. TSP and B suggest similar approaches to idempotence, but failed to abstract out the notion of just needing idempotent code. Here, we abstract out the need of idempotence for lock-free locks and suggest a very different, as well as more efficient and general, approach to achieving idempotence. We also point out that to nest locks, we simply need the locking code itself to be idempotent, leading to locking code that is very simple.

In our approach to idempotence, instead of using the context saving of TSP-B, we maintain a shared log among processes running the same code. The log keeps track of all reads from shared mutable locations, as well as some other events, such as memory allocations. Whenever the code executes a loggable operation, it commits it to the log using a compare-and-swap (CAS). Among the processes running the same code, whichever commits first, wins. All others take the value committed instead of their attempted commit. In this way, they all see the same committed values, e.g., the same reads, even though they are running in an arbitrary interleaved manner.

One key advantage of our approach is that the user can write concurrent algorithms based on fine-grained locks, and then either run them entirely in a *lock-free mode* (with helping) or a *blocking mode* (no helping). The blocking mode can use a standard lock implementation without logging. The helping mode will log, at some additional cost, but guarantee lock-free behavior. Another key advantage over TSP-B is that our approach is based on try locks, instead of strict locks, which turns out to be important for the efficiency of optimistic use of fine-grained locks.

We have implemented our approach as a C++-based library called Flock. Based on the library we have implemented several data structures based on try-locks, including singly linked lists, doubly linked lists, binary trees, balanced blocked binary trees, (a,b)-trees, hash tables, and adaptive radix trees (ART). We compare the performance of our versions in lock-free mode and blocking mode to the most efficient existing data structures we found, both lock-based and lock-free. The lock-based data structures generally perform slightly better under controlled environments with one process per processor, but perform significantly worse when oversubscribing with multiple processes per processor. Comparing running our algorithms in lock-free vs. blocking mode, the lock-free performance rarely has more than 10% overhead, and typically much less. However, with oversubscription, the lock-free mode greatly outperforms the blocking mode by up to 2.4x.

Our contributions include the following:

1. We present a new practical approach to achieving idempotence in general code, which relies on logging rather than context saving.
2. We present a new approach to lock-free try-locks. They can be nested.
3. We develop a general library-based interface to support our ideas.

4. We compare several existing approaches with ours, both using locking and without using locking.
5. We develop the first direct lock-free implementation of adaptive radix trees.

3.1.1 Example of Using Lock-Free Locks

To be concrete on how lock-free locks are used in our framework, we give an example of maintaining a concurrent sorted doubly-linked list supporting insert, delete, and find. The example uses optimistic fine-grained locks [99, 100]. Our C++ code using Flock[1] is given in Algorithm 3.1. Each node holds a key and value, previous and next pointers, a lock, and a flag indicating whether the node has been removed. The `flck::atomic` wrapper around `next`, `prev`, and `removed` (lines 2–4) indicates that these are shared mutable values. They need to be read using a `load`, with a similar interface to a C++ `std::atomic`. Flock will log loads of mutable values when inside a lock. The key and value fields are immutable so they need not be put in `flck::atomic`.

Locks are attempted with the `try_lock` function. It takes a lock as an argument, as well as a *thunk* (a function with no arguments). In Flock, the thunk is simply a C++ lambda expression (which is essentially an anonymous function) containing the code to be run when the lock is acquired. If the lock is free, `try_lock` acquires the lock, runs the thunk, releases the lock, and returns the thunk's return value (a boolean). Otherwise it returns false. The `try_lock` function forces locks to be properly nested. This is important for our lock-free locks since the thunk captures the code that might need to be helped by another `try_lock`. In Section 3.3 we describe a function that avoids pure nesting and supports, for example, hand-over-hand locking.

The `find_node` finds the first node with a key greater than or equal to the requested key. It requires no locks. The `find` just extracts the value from the node if the key matches.

The `remove` first finds the node `n` potentially containing the key. If it does not contain the key, then it returns `false` indicating the key was not in the list. Otherwise it tries to acquire a lock on the previous node (`prev`) and `n`. If either fails because they are already locked, the condition on line 39 will be false and the `while` loop will repeat. The conditions on lines 41 and 42 validate that the previous node has not been deleted, and `prev->next` still points to `n`. If either test fails then the `while` loop is repeated. If the tests pass, the code in the lock loads the next pointer from `n`, marks `n` as removed, splices it out of the doubly linked list, and retires its memory[2]. Note that a lock is not required on the node pointed to by `next`. This is because a deletion of `next` or an insertion of an element before `next` would require a lock on `n` so it cannot happen concurrently. The `insert` is similar to `remove`.

This locking-based code for doubly-linked lists is much simpler than any lock-free versions we know of [9, 20, 77, 148, 155]. The difficulty in generating a lock-free version based on CAS is that lines 46–47 need to be applied atomically, as do lines 29–30. Our approach gives us a lock-free algorithm using the simple lock-based algorithm. As we show in our experiments, the

[1]We use the abbreviation `flck` in our code as `flock` is already reserved in C++.

[2]Flock uses an epoch based memory manager. The `retire` puts the pointer aside and frees the memory it points to when it is safe (after all concurrent operations finish).

```
struct node {
  flck::atomic<node*> next;
  flck::atomic<node*> prev;
  flck::atomic<bool> removed;
  Key k; Value v; flck::lock lck;
  node(Key k, Value v, node* next, node* prev)
    : k(k), v(v), next(next), prev(prev), removed(false)
    {};};

node* find_node(node* head, Key k) {
  node* n = (head->next).load();
  while (k > n->key) n = (n->next).load();
  return n;}

std::optional<Value> find(node* head, Key k) {
  node* n = find_node(head, k);
  if (n->key == k) return n->value; // found
  else return {}; }                    // not found

bool insert(node* head, Key k, Value v) {
  while (true) {
    node* next = find_node(head, k);
    if (next->key == k) return false; // already there
    node* prev = (next->prev).load();
    if (prev->key < k && prev->lck.try_lock([=] {
          if (prev->removed.load() || (prev->next).load() != next) // validate
            return false;
          node* newl = allocate<node>(k, v, next, prev);
          prev->next = newl; // splice in
          next->prev = newl;
          return true;}))
      return true;}}; // success

bool remove(node* head, Key k) {
  while (true) {
    node* n = find_node(head, k);
    if (n->key != k) return false; // not found
    node* prev = (n->prev).load();
    if (prev->lck.try_lock([=] {
          return n->lck.try_lock([=] {
            if (prev->removed.load() || // validate
                (prev->next).load() != n)
              return false;
            node* next = (n->next).load();
            n->removed = true;
            prev->next = next;  // splice out
            next->prev = prev;
            retire<node>(n);
            return true;});}))
      return true;}} // success
```

Algorithm 3.1: Sorted doubly-linked lists using fine-grained optimistic locks with FLOCK. FLOCK code shown in red.

lock-free version is almost as fast as the locking one without oversubscription, but much faster with oversubscription.

3.2 Idempotence

To achieve lock-free critical sections, processes must be able to help each other. In particular, if some process holds a lock and crashes, others must be able to release the lock. Since it is possible that the crashed process has already begun its critical section, the other processes must complete its critical section for it before releasing the lock.

This leads to the need to have *idempotent* critical sections. Intuitively, a piece of code is idempotent if, when it is executed multiple times, it only appears to take effect once. Thus, if we have idempotent critical sections, processes can safely help execute someone else's critical section, without worrying about who else has also executed it. Some code is naturally idempotent. For example, a critical section that contains just one CAS instruction, which does not suffer from the ABA problem, is idempotent. After it is executed for the first time, subsequent executions of it would have their CAS fail, thus leaving the memory in the same state. Many hand-designed lock-free data structures achieve their lock-freedom by allowing helping in such short, naturally idempotent sections.

In general, however, most code is not idempotent. For example, code incrementing a counter would yield different resulting counter values if it is executed several times. Thus, general lock-free constructions must be able to make general code idempotent. Several approaches in the literature have shown how to do so [14, 17, 18, 159]. In this section, we define idempotence formally and present a new construction that makes any piece of code idempotent.

3.2.1 Idempotence Definition

A *thunk* is a procedure with no arguments [90]. Note that any procedure with given arguments can be made a thunk by wrapping it in code that reads its arguments from memory. The pseudocode we present is specialized for thunks returning true or false but it can be generalized to any return type.

We follow the definition of idempotence introduced in [17]. Let T be an instance of a thunk. A *run* of T is the sequence of steps on shared data taken by *a single process* to execute or help execute T. The runs of T by different processes can be interleaved and each run may take a different branch through T depending on the memory state that it sees. A run is *finished* if it reached the end of T. We say a sequence of steps S is *consistent* with a run r of T if, ignoring process ids, S contains the exact same steps as r. We use $E \mid T$ to denote the result of starting from an execution E and removing any step that does not belong to a run of the thunk instance T.

Definition 1 (Idempotence [17]). *A thunk is idempotent if all instances of it are idempotent. An instance of a thunk T is idempotent if in any valid execution E consisting of runs of T interleaved with arbitrary other steps on shared data, there exists a subsequence E' of $E|T$ such that:*

1. *if there is a finished run of T (response on T), then the last step of the first such finished run must be the end of E',*

```
type Log = shared<entry>[logSize];
type Thunk = function with no arguments returning bool

private process local:
  Log* log; // the current log for a process
  int position; // the current position in the log

struct descriptor:
  Log* log;
  Thunk thunk;
  flck::atomic<boolean> done;
```

Algorithm 3.2: Types and global variables used in Algorithm 3.3.

2. *removing all of T's steps from E other than those in E' leaves a valid execution consistent with a single run of T.*

Intuitively, this definition allows an instance of a thunk T to be executed by several processes (in several runs of T), but other than one copy of each step executed for T, the rest are not effectual (i.e. have no impact on the rest of the execution). Furthermore, after one run of T completes, no other runs of T can execute an effectual step.

We assume that a thunk may have *thunk-local* memory which can only be accessed by processes executing the thunk. In our simulation the log is thunk-local. Such memory is not "shared data" as defined in Definition 1.

3.2.2 Our Approach to Idempotence

We now present a new approach to achieving idempotence in any code that is ABA-free. We note that it is easy to make code ABA-free by attaching a counter to any memory location that suffers from the ABA problem, and updating that counter every time the value is updated (our implementation does this). Rather than basing our idempotence construction on *context saving*, as were previous general idempotence constructions, we base our approach on using a shared *log*. We present pseudocode for the approach in Algorithms 3.2 and 3.3. For memory management, it assumes a *sysAlloc* primitive which returns an unused block of memory and a *sysRetire* primitive which delays freeing the memory block until it is safe.

We store each instance of a thunk in a struct, called the *descriptor*. The descriptor also stores a pointer to the log associated with the thunk instance as well as a boolean indicating whether or not the instance has already been executed. The log keeps track of all values read, allocated or retired while executing the thunk instance. The `shared<T>` type indicates a variable of type `T` that is shared among processes. The log, however, is *thunk-local*; any process executing this thunk instance uses the same descriptor struct, so the log is shared by all processes that execute this thunk instance,[3] but no other process can access this log.

We implement five operations for idempotent code: *load*, *store*, *CAM* (a CAS that does not return any value or any indication if it succeeded or failed), *allocate*, and *retire* using

[3]Note that this differs from distributed logs used in, for example, optimistic transactional memory [100], where each process has its own log. It also differs from logs used to commit successful transactions.

```
descriptor* createDescriptor(Thunk f):
  Log* log = allocate<Log>();
  return allocate<descriptor>(log, f, false);

void retireDescriptor(descriptor* T):
  retire<Log>(D->log);
  retire<descriptor>(D);

bool run(descriptor* D):
  Log* old_log = log; // store existing log and position
  int old_pos = position
  log = D->log; // install D's log
  position = 0;
  bool returnVal = D->thunk(); // run thunk
  log = old_log; // reinstall previous log and position
  position = old_pos
  return returnVal;

<V, bool> commitValue(V val):
  if (log == null): return <val, true>;
  bool isFirst = log[position].CAS(empty, val);
  V returnVal = log[position].read();
  position++;
  return <returnVal, isFirst>;

struct flck::atomic<V>:
  shared<V> val;
  V load():
    V v = val.read();
    return commitValue(v).first;
  void store(V newV):
    V oldV = load();
    val.CAS(oldV, newV);
  void CAM(V oldV, V newV):
    V check = load();
    if (check != oldV): return;
    val.CAS(oldV, newV);

V* allocate<V>(args):
  V* newV = sysAllocate<V>(args); //use system allocator
  <obj, isFirst> = commitValue(newV);
  if not isFirst: sysFree<V>(newV);
  return obj;

void retire<V>(V* obj)
  <_, isFirst> = commitValue(1);
  if isFirst: sysRetire(obj);
```

Algorithm 3.3: Idempotent primitives. The **entry**s of the log are assumed to hold any type that fits in a word (or two if using double width CAS). The log is of fixed sized, but could grow by adding blocks as needed (see Section 3.4 for details).

the memory primitives *read*, *write*, *cas*, *sysAlloc* and *sysRetire*. Any thunk implemented from these instructions can then be run idempotently. For ease of use, load, store and CAM are implemented in a struct called `flck::atomic` that can wrap any type. This is modelled after C++'s `std::atomic` type, which is used for shared mutable variables. Any variable declared as `flck::atomic` automatically uses our idempotent versions of the corresponding primitives, keeping programmer effort to a minimum. We assume that CAMs and stores do not race on the same location. For our purposes a value is *non-mutable* (constant) if it is written once (e.g. on initialization) and only read after it is written. Any non-mutable value, or any local variables/locations can be read and written as usual without using a `flck::atomic`.

The idea of the approach is that each process keeps track of its current log (line 5) and how many items it has logged in it so far while running the corresponding thunk instance (its *position*, on line 6). Thus, when it starts executing a new thunk instance, it initializes its position to 0 and its local log to point to this thunk instance's log (lines 24 and 23). The process saves its previous log and position so that it can go back to its previous thunk instance when it finishes executing the new one. This is useful for executing nested thunks. Once a process has installed its new log and initialized its position, it can start running the thunk instance. Whenever it executes a new loggable instruction (load, allocate or retire), it uses the shared log of the thunk instance to record the return value of this instruction and to see whether others have already logged it.

Values are stored in the log using a helper function called *commitValue* (line 30). This function takes in a value to be logged; intuitively, this is the intended return value of the current instruction. The process uses its current position to index into its thunk instance's log. It tries to commit its value by using a CAS on `log[position]`, with old value `empty`, and new value equal to the value it would like to log. All log entries are initialized to `empty` and we assume that no process attempts to write `empty` into a `flck::atomic` variable. The process then checks what value is written in `log[position]`, and returns this value, as well as a boolean indicating whether or not its CAS succeeded (i.e. whether it was the first to execute this instruction on this thunk instance). When the process does not currently have a log (i.e. is not currently executing a thunk), the `commitValue` function simply returns the input value and the success flag set to true (line 31). With our locks this happens when the instruction is executed outside of all locks. For example, no logging is needed for the loads on line 12 in Algorithm 3.1 since they are not in a lock, but the load on line 41 is logged in the descriptor for its surrounding lock.

To load a value from a given `flck::atomic` variable, a process simply does a read from the variable, and then tries to commit its value to the log by calling `commitValue`. `commitValue` returns the value that was successfully committed which is in turn returned by the load (line 33). In this way, the process returns the same value from its load as any other process executing this load for this thunk instance.

To store a value in a given `flck::atomic` variable, the process first executes a load as described above, thereby logging the value present before the store occurred, or discovering what that value was (if this store was already executed by a different process). The process then executes a CAS with expected value equal to the value returned from the load. Recall that we assume that shared memory locations are ABA free, and therefore this ensures that all CAS attempts but the first will fail. The CAM operation works similarly to the store, but with an additional check to make sure the value returned by the load matches the expected value. It only executes a CAS if this is the case. By performing a load before the CAS, we guarantee that the expected value was stored in the memory location at some point. Combined with the ABA-free assumption, this prevents a potentially dangerous scenario where the expected value is written into the memory location after the CAS, causing future executions of the CAM to potentially succeed and no longer be idempotent. It is important that the CAM does not return the return value of its CAS, since this value could be different for different processes that execute it, and could therefore violate idempotence (externalize a different result). An example of use case for this idempotent CAM will appear in Algorithm 5.5 of Chapter 5.

We also provide allocate and retire operations for idempotence. The idea is again to use the thunk log to commit values. To allocate a new object, the process allocates this object using the system-provided allocation mechanism, and then uses `commitValue` to install this new object in the log. If it is the first to do so, then the allocation is done, and this new object is returned. Otherwise, the process destroys its newly allocated object, and instead returns the object that was already installed in the log.

To retire an object, the processes use the log to compete for 'ownership' of this object. The first process to commit a boolean retirement flag on the log is responsible for retiring this object. All other processes simply skip retiring it if they discover, by trying to commit a flag to the log, that some other process already owns this object. In this way, each object is retired at most once. Standard garbage collection techniques can then be used to collect retired objects when it is safe to do so.

The commitValue can also be used directly by the user to commit the result of any non-deterministic instruction. For example, if there is an instruction that generates a value based on random noise in the processor, this needs to be committed so all instances of the thunk agree on it.

We now show that our idempotence construction is correct; that is, the `flck::atomic` type implemented in Algorithm 3.3 is linearizable, and any thunk that wraps all the mutable shared variables it accesses in a `flck::atomic` type is idempotent. We begin by outlining a proof of idempotence. For the following theorem, we relax Definition 1 so that retire operations in E' are allowed to appear later than they would have in a single run of T. This has no effect on correctness and at worst it delays the reclamation of memory. Our idempotence construction requires this relaxation because a process can go to sleep before peforming the `sysRetire` on line 58, and in the meantime, other processes can perform future operations of the thunk instance, making the retire appear out-of-order.

Theorem 3.2.1. *Replacing each mutable shared variable accessed by a thunk T with a* `flck::atomic` *type and allocating and retiring all objects in T with the provided allocate and retire operations yields an idempotent version of T.*

Proof. We begin with a brief outline. The idea is that all processes running the same thunk instance will stay synchronized in the sense they will have the same state at the same point of their execution. Whichever gets to a loggable event first will log it, and all others will see it is already logged and use the same value. In this way, they all see the same values, and stay synchronized. It also means their position in the log will be synchronized. Memory allocation and retiring is safe since only the first run of each allocate will keep its allocated value and only the first run of each retire will retire the value. For stores and CAMs, only the first such operation will succeed and all others will fail, because of our ABA-free assumption. Therefore only the first will be visible. This argument is described in more detail below.

Given an execution E consisting of runs of T interleaved with arbitrary other steps on shared data, we will construct a subsequence E' of $E|T$ that satisfy the criteria from Definition 1 (with the relaxation that retire operations in E' are allowed to appear later than they would have in a single run of T). Throughout the proof, we will refer to load, store, CAM, allocate, retire as operations, and executions of primitive shared memory instructions such as read, write, and CAS as steps.

We begin by viewing the execution at the level of operations. We show by induction that all runs of T execute the same sequence of operations with the same arguments and return values. As the base case, note that all processes that execute T start with the same local variables, and T takes no arguments. Therefore, they begin the execution in the same state. As the inductive hypothesis, assume that the first $k-1$ operations executed by T are the same across all runs and have the same arguments and return values. Consider the kth operation O_k. Since all previous operations returned the same value across all runs, then O_k is the same operation and is called with the same arguments across all runs. Note furthermore that if O_k executes line 32, the CAS on that line is successful in exactly one instance. All processes executing O_k use position k to access the log, and no process executing a different operation uses position k. Therefore, before the first execution of line 32 for O_k, `log[k] = empty`. Since we assume `empty` is never written in any allocated variable, the new value of the CAS on line 32 will never be `empty`. Therefore, the first instance of that CAS will be successful, and all others will fail. If O_k is a `load` or an `allocate`, since those operations return the value read from `log[k]` after the first CAS on line 32 for O_k, all its instances will return the same value. Note that all other operations do not return a value, so the claim holds.

Next, we construct the subsequence E' by picking steps so that each operation O_k appears to only run once. We will ensure that all steps picked from runs of O_k appear before those picked from runs of O_{k+1}, except when O_k is a `retire` operation in which case its call to `sysRetire` may appear later. For each operation O_k, consider the run that executes the CAS on line 32 first. We pick a prefix of that run, starting from the beginning of O_k up to when it executes line 32 (inclusive), to be part of E'. As shown in the previous paragraph, executions of the CAS on line 32 by other runs of O_k will return false. Next, we pick the first execution of line 33 by any run of O_k to be part of E', and we pick the remaining steps differently depending on what type of operation O_k is. `load` operations do not perform any more shared memory steps so we are done. Let r be a run of O_k that is consistent with the sequence of steps we have picked so far for E'. Since we picked the successful instance of line 32, `isFirst` is set to true for r. Therefore, if O_k is an `allocate`, then r will not execute any more shared memory steps after line 33, so E' contains all of r's steps. If O_k is an `retire`, then whichever run executed the successful CAS on line 32 will eventually execute a `sysRetire` on line 58, and we pick that `sysRetire` to be part of E' (if it exists in E). Note that this `sysRetire` may appear in E' after steps by future operations and this is allowed by our relaxed idempotence definition. If O_k is a `store`, then we pick the first execution of the `CAS` on line 44 to be part of E'. All executions of this CAS by future runs of O_k will return false because `oldV` was previously stored in `val` and we assume `flck::atomic` types are ABA-free. Finally, suppose O_k is a `CAM`. Since the value of `check` on line 47 was read from the log on line 33, all runs of O_k will have the same value for `check`. Therefore, either all runs will execute the `CAS` on line 48 or none of them will. If they execute the `CAS`, then we pick the first such step to be part of E', just like for `stores`. Otherwise, O_k performs no more shared memory steps and we are done.

Picking steps in this manner ensures that T appears to run once in E' and if there is a finished run of T in E, then the last step of the first finished run will be the end of E' (with the exception of `sysRetire`). Furthermore, the steps in $E|T$ that we did not pick have no effect on shared memory so removing them still leaves a valid execution. This is the case for any removed CAS operation because they all return false. Also, memory locations allocated by removed `sysAllocate` operations are never used since they are never committed to the log. Finally, the `sysFrees` that were removed correspond to the removed `sysAllocate` operations. Therefore, removing all of T's steps from E other than those in E' leaves a valid execution.

□

Idempotent by itself does not guarantee that we are not over-allocating or double freeing. To prevent memory leaks, every block of memory allocated on line 51 that does not get committed to the log is freed on line 53. We also use the shared log to ensure that each object is retired no more than once on line 58.

To complete the correctness proof, we also need to show that `load`, `store`, and `CAM` are linearizable in executions where each instance is run only once. Intuitively, this is because in the absence of repeated runs, the `load` operation simply reads and returns the variable `val`, and the `store` and `CAM` operations simply read `val` and try to update it with a CAS. This is a well-known linearizable implementation of `load`, `store`, and `CAM` (also works for `CAS` instead of `CAM`) using just `load` and `CAS`, and it is linearizable as long as `stores` and `CAMs` do not race.

As mentioned, most previous approaches to idempotence have been based on *context saving* [14, 17, 18, 30, 159]. This involves storing out a program counter and current state of all local variables at important events (e.g. shared memory operations), and possibly loading and installing a new context if already stored. Our approach never needs to store a program counter or local state since the processes are running “synchronously” and have the same local state. For large thunks, and frequent helping, however, our method potentially does have an additional cost. In particular, we always start helping from the beginning of a thunk while the other methods will start at the point of the last context saved by any process. Our method is therefore particularly well suited for short thunks, which is the intended use with fine-grained locks, and possibly not as well suited for long running thunks.

3.3 Lock-free Locks

We now describe how we implement a tryLock. It is important that tryLocks can be nested to allow a process to hold multiple locks at the same time. This means the locking mechanism itself must be idempotent or otherwise safe to use when there are multiple threads helping to acquire the lock. In particular, consider an operation O_1 that takes an outer lock L_a and inside the lock takes an inner lock L_b. If another operation O_2 encounters L_a locked, it will help O_1

```
struct taggedDescr :
   descriptor* d;
   bool isLocked;

type Lock = flck::atomic<taggedDescr>;

bool runAndUnlock(Lock* lock, taggedDescr descr):
  bool result = run(descr.d);
  descr.d->done.store(true);
  lock->CAM(descr, taggedDescr(descr.d, false));

bool tryLock(Lock* lock, Thunk $f$):
  bool result = false;
  taggedDescr currentDescr = lock->load();
  if (not currentDescr.isLocked) :
    descriptor* myDescr = createDescriptor($f$);
    taggedDescr myTaggedDescr = {myDescr, true};
    lock->CAM(currentDescr, myTaggedDescr);
    currentDescr = lock->load();
    if ((myTaggedDescr.d->done).load() or
         myTaggedDescr == currentDescr) :
      result = runAndUnlock(myTaggedDescr); //run self
    else if (currentDescr.isLocked) :
      runAndUnlock(currentDescr); // help other
    retireDescriptor(myDescr);
  else : runAndUnlock(currentDescr); // help other
  return result;

void unlock(Lock* lock):
  taggedDescr currDescr = lock->load();
  lock->CAM(currDescr, taggedDescr(currDescr.d, false));
```

Algorithm 3.4: Idempotent TryLock

execute its critical code. This means it will help O_1 acquire L_b and, if successful, run the code of O_1 in that lock. Note that the outer most tryLock in a sequence of nested tryLocks need not be idempotent because it will never be helped by another process.

Based on our technique for idempotence, it turns out to be quite simple to implement the locking mechanism so that it is idempotent. Our code is given in Algorithm 3.4. It uses a type called `taggedDescr`, which is a pair consisting of a boolean and a pointer to a descriptor. The boolean indicates whether or not the lock is currently taken. It is easy to put these two fields into a single word by stealing a bit from the pointer. A `Lock` is then defined as a `flck::atomic<taggedDescr>`.

An attempt at acquiring the lock starts by reading the lock and checking if it is currently locked. If not locked, the algorithm creates a descriptor for the thunk instance f (line 16) and tags it to mark that it is locked (line 17). It then attempts to install the descriptor on the lock using a CAM (`flck::atomic` does not support CAS). Since the CAM does not return whether it succeeds, the algorithm needs to read the lock again (line 21) to check if the lock was successfully acquired for its descriptor. If acquired or if previously acquired and now done, it runs the code and releases the lock (line 22). If not acquired but `currentDescr` is locked, then the algorithm helps the descriptor on the lock and unlocks it (line 24). Whether the CAM was successful or not, `myDescr` needs to be retired (line 25). If on line 15 the lock is already locked, then the algorithm helps the descriptor on the lock and unlocks it after finishing helping (line 26). Finally the `result` is returned, which will only be true if the lock was successfully acquired and the thunk f returns true.

We now argue correctness. We say a tryLock is correct if it either fails, in which case none of the critical code (thunk f) is run and it returns false; or it succeeds, in which case all its critical code is run and the tryLock returns its value. If successful, no other critical code on the same lock can run concurrently. By this definition, the tryLock could always fail, but this would not satisfy progress bounds, and in particular for us, our lock-free bounds. We say a successful tryLock *enters* on the step the lock is changed to point to the tryLock's descriptor and *exits* on the step when the lock is changed from locked with its descriptor to unlocked.

Theorem 3.3.1. *The tryLock in Algorithm 3.4 is correct as long as* `run(descriptor)` *runs the user code in the thunk f idempotently, and the operations on a Lock (*`load`, `CAM` *and* `store`*) and on descriptors (*`createDescriptor` *and* `retireDescriptor`*) are idempotent.*

Proof. (Outline). The code in a thunk consists of the user level code and possibly the code of one or more nested tryLock. Together this is idempotent by assumption.

In the algorithm, a descriptor is run if and only if the tryLock enters and the lock is set. The descriptor is run by the `runAndUnlock` method which can be called on line 22 by the process that installed the descriptor, or on lines 24 or 26 by the helping processes. Some process (either the primary process or a helper) will finish the thunk first. Since the thunk is idempotent, any processes working on the same descriptor after that point will have no effect. The lock is only released after the thunk is first finished so the code can only have an effect between when the successful tryLock enters and exits. Since there is a unique descriptor on the lock during this time, no other thunk on the same lock can appear to run concurrently. There could be leftover thunks from earlier successful attempts on the lock, but they will have no effect.

If either the lock was already taken on line 14 (i.e. the check on line 15 fails) or the attempt to install a descriptor was unsuccessful on line 18 (i.e. the check on line 21 fails), then the tryLock fails and returns false. Otherwise, its descriptor was successfully installed, and it returns the result of running that descriptor on line 22. Note that it is important to check the descriptor's done flag on line 20 because even when the descriptor is successfully installed on line 18, the `load` on line 19 might not see it because it might have been helped and replaced by another process. Checking the done flag ensures that the tryLock will always return the return value of the descriptor it installed if its CAM on line 18 is successful. □

The theorem does not depend on a particular implementation of idempotence, but works with ours since ours satisfies the specified conditions.

We now show that tryLocks are lock-free. For this purpose we make some assumptions. Firstly, we assume the locks have a partial order $<_\ell$, and that when nesting locks they are acquired in decreasing order. This is a relatively standard assumption for lock-based algorithms since it prevents lock-cycles and deadlock. Secondly, we assume that each tryLock includes at most one other tryLock directly inside of it. Note that this still allows arbitrary depth of nesting since the one inside can itself contain another lock inside it.[4] We refer to locks that satisfy these two conditions as *simply nested* and we refer to the outer most lock as the *top-level* lock. We say that a simply nested tryLock *recursively succeeds* if it acquires its lock as well as all locks nested inside of it and finishes executing its critical section. Note that for a tryLock a if any one tryLock nested in it recursively succeeds then they all do, including a. We say a tryLock succeeds if it acquires its lock and finishes executing its critical section.

We also need to bound the time of user code in a lock, otherwise helpers could never complete helping. We defined *step count* for a tryLock recursively by counting the number of steps performed by the thunk passed to the tryLock as follows. We count all functions in the idempotent interface as unit cost plus the cost of any user code inside of them—in particular `sysAllocate` and `sysRetire` count toward user code. We count a nested `try_lock` as unit cost plus the step count of the code in its critical region.

Theorem 3.3.2. *Consider an algorithm using simply nested tryLocks for which the maximum step count for any tryLock, not including helping, is bounded. In such an algorithm, a tryLock, including any helping it does, will run in a bounded number of steps, and for every bounded number of tryLock attempts at least one top-level tryLock will recursively succeed.*

Proof. We say that a tryLock b *directly helps* another tryLock c if (1) line 24 or line 26 of b runs the thunk installed by c or (2) `currentDescr` on line 19 of b is unlocked and belongs to c. In the second case, b does not actually help c, but c must have acquired its lock after line 14 of b and released it before line 19 of b so we can give b credit for helping. Importantly, by this definition, a tryLock either succeeds or *directly helps* another tryLock. We say that a tryLock *a helps c* if a or a tryLock nested in a directly helps c. We say b *recursively helps* c if either b helps c or b helps another tryLock that recursively helps c. Every tryLock helps at most one other due to idempotence and the nesting property that we assume. More specifically, if a nested tryLock

[4]We expect this requirement is not necessary, but our proof relies on it and it is true for all our tryLock-based data structures.

fails, no new locks can be taken until the top level tryLock completes. This implies that helping forms a chain where the next tryLock in the chain is helped by the previous one.

Now note that if a helps b due a conflict on lock l_1 and b helps c due to a conflict on lock l_2, the tryLock that attempts l_2 is nested inside the tryLock that acquired l_1. Since nested locks are acquired in the partial order $>_p$, we have that $l_1 >_p l_2$ and more generally that locks decrease along any helping chain. Furthermore, each tryLock along the helping chain is owned by a different process. To see why, we first note that if a directly helps b, then b installed its descriptor before the end of a since a must have seen b's descriptor. Suppose p is the process that owns a. All the locks recursively helped by a have lower priority and their descriptors were installed before the end of a, so a cannot recursively help a tryLock owned by p unless the locking order is violated. Therefore, the same process will not appear twice along any helping chain, so the chain will have bounded length (at most P) and will end with a recursively successful tryLock. Since we assume the user code takes a bounded number of steps, each instance of helping along the helping chain also takes bounded steps. Hence, running any tryLock takes bounded steps (including all recursive helping along the chain). Furthermore, since there are a bounded number of locks on the chain, the number of tryLocks responsible for completing the last one is also bounded. Finally we note that although the last one might not be top-level, the fact it recursively succeeds implies the top-level tryLock that contains it recursively succeeds. □

This theorem indicates that simply nested tryLocks are lock-free in that a top-level tryLock must recursively succeed in a finite number of steps. Recall that this implies all the tryLocks nested inside it are successful as well. The theorem does not, however, imply wait-freedom since a particular process could continuously fail to acquire a lock. It also does not, by itself, guarantee an algorithm using simply nested tryLocks is lock-free. In an algorithm based on optimistic fine-grained locks, for example, we might need to retry not because a lock failed to be acquired, but instead because the data structure changed between our optimistic traversal and our acquiring of the locks (e.g. the consistency check on line 41 of Algorithm 3.1). In all the algorithms we consider, however, a change in the data structure means another operation has made progress. In the `remove` from Algorithm 3.1, for example, the consistency check can only fail if in between the `find_location` and when the lock on `cur` is acquired, either (1) `cur` is deleted or (2) it is updated to point to a new `next`. In either case, the algorithm has made progress by completing an operation. A similar argument can be made for the `insert`. Therefore the ordered list algorithm based on our tryLocks is lock-free, as are the other algorithms we consider.

It can be useful to release a lock early before the scope of the thunk associated with the acquired lock completes. We supply a `unlock` for this purpose. It takes a lock that is currently acquired by the thread and unlocks it. Its behavior is undefined if the thread has not acquired the lock. As mentioned in the introduction, this can be used for hand-over-hand locking (also called lock-coupling) [16].

The code for tryLock can be modified to support a strictLock that always acquires the lock before returning, by first creating the descriptor, and then putting the attempt to acquire a lock into a while loop. We have implemented an optimized version of such a strictLock and compare it to the tryLock in Section 3.6. We note that this implementation of strict locks is not simply nested so is not covered by Theorem 3.3.1. However, it should be possible to adapt TSP's proof [159] to show that strict locks are lock-free.

3.4 The Flock Library

We have implemented a C++ library, Flock, based on our lock-free locks approach. This approach does not require any C++ specific features and can be implemented in other languages as well. It supports a `flck::atomic` wrapper to use on any shared variables that can be mutated inside a lock. Note that these variables can also be accessed outside of critical sections and no logging will be performed in that case. Our library also supplies a `lock` type and a `try_lock` function. The `flck:atomic` wrapper has a similar interface to the C++ `std::atomic` wrapper. In particular, it supports `load`, `store` and `cam`. The assignment operator (=) is overloaded so that it becomes a call to store. Flock also supports `allocate` and `retire` which are integrated with its epoch-based collector. An example of how to use Flock is given in Algorithm 3.1. The library is available at `https://github.com/cmuparlay/flock` [5].

Here we discuss several specifics about the implementation, including some optimizations.

Epoch-based collection. Flock uses an epoch-based memory manager [71, 79] for simplicity and it is compatible with other concurrent memory management schemes as well. In an epoch-based memory manager, each operation runs in an epoch, each of which is associated with an integer that increases over time. Managing memory with epochs requires some additions to the the implementation of idempotent code. In particular, when a thread helps another thread, it is taking on the responsibility of that other thread. It therefore needs to also take on the other thread's epoch number. This is because the other thread can complete its operation before the helping thread finishes its helping steps. To implement this, when Flock has to help inside of a `try_lock`, it changes its epoch to be the minimum of its epoch and the epoch of the thunk it is helping. Note that choosing the minimum prevents the reclamation of any object retired during either epoch. When it is finished helping, it restores its epoch to what it was before helping. The descriptors are also allocated and retired with the same epoch-based collector, with one optimization. In particular if a descriptor is never helped, which is the common case, then it can be reused immediately instead of being retired. To implement this, we keep a flag on the descriptors which is set when helping. This requires some careful synchronization.

ABA. Although the idempotent implementation in Algorithm 3.3 requires that mutable shared variables are ABA free, an `atomic` in Flock does not have this requirement. To allow for this, Flock keeps tags on mutable locations. A simple implementation is to use a 64-bit counter, and increment the counter on each update. Assuming mutable values can be up to 64-bits, this can be implemented with double-word (128-bit) loads and CASes. Unfortunately double-word loads are particularly expensive on current machines. Flock has two optimizations to avoid them, one which supports 64-bit values, and one for 48-bit values, which is sufficient for a pointer.

The first optimization still uses a 64-bit counter on every `flck::atomic`, but avoids any double-word loads. A key observation is that a `load` only needs to log the value, and therefore

[5]Our paper on Flock [21] originally referred to `flck::atomic` as "`mutable_`", but the newest version of the Flock library uses `atomic`.

only needs to read this value (rather than both the value and the counter). Another observation is that a `store` (or `cam`) does not need to read the counter and value atomically. Instead, it can first read the counter and then the value, followed by a double-word CAS to the `flck::atomic`. This is safe since the value can only change if the counter changes.

The second optimization avoids the extra 64-bit counter on each mutable location and any double word operations. Instead it uses a safe lock-free approach that only requires a 16-bit tag. It uses an announcement array to ensure that wrapping around is safe—i.e., it never uses a tag that is announced. All the experiments in Section 3.6 use this version since the mutable shared variables are no larger than a pointer.

Constants and Update-once Locations. Shared, constant locations do not need to be wrapped in an `atomic` and can just be read directly. A constant location is one that is written once and only read after it is written. The write could happen during construction of the object that contains it or after. For example the key and value in the list link in Algorithm 3.1 are constants. Flock also supports update-once locations. These are locations that have an initial value, and are updated at most once. Reads can happen before or after the update. The `removed` flag in a link in Algorithm 3.1, for example, is updated once. Update-once variables are ABA free and therefore do not need a tag. Furthermore, the `store` can be implemented with a simple write instead of a `load` and then a `CAS`. This is because only the first such write will have an effect. The other writes by helping threads will write the same value and have no effect.

Arbitrary Length Logs. In general it cannot be determined ahead of time how long a log will be. Flock therefore implements logs that can dynamically increase in size. In the implementation, a log has a fixed block size (7 by default). If it runs out, another block is allocated. To do this idempotently, the first thread that runs out allocates the block and attempts to CAS it into a next-block pointer. If it fails, it frees its block and takes on the block that succeeded.

Avoiding CASes. We found that one of the most expensive aspects of helping is contention due to CASes on both the log and mutable locations. This is especially true under high contention when there is a lot of helping. To significantly reduce this contention we use a compare-and-compare-and-swap. In particular, before doing a CAS, the location is read and compared against the expected value, and if not equal the CAS can be avoided. When helping under high contention it is often not equal (someone else already executed the CAS) so many of the CASes are avoided. This rather simple change made a significant improvement in performance under high contention—sometimes a factor of two or more.

Capturing by Value. In the code in Algorithm 3.1, one might notice the "`[=]`" in the definition of the lambda's. This indicates that all free variables in the lambda defined outside of it are captured by value, as opposed to by reference—i.e., they are copied into the thunk. This is important since the lambda might outlive its context, and any surrounding stack allocated values could be destructed while being helped. Indeed if the `[=]` is replaced by `[&]` (by reference), Algorithm 3.1 would be incorrect—for example, the variable `prev` on line 38 could be reused while the lambda is being helped.

3.5 Data Structures

We have implemented several concurrent data structures using Flock. These data structures include the doubly linked list described in Section 3.1.1 (`dlist`), a singly-linked list [83] (`lazylist`), an adaptive radix tree [103, 104] (`arttree`) which is a state-of-the-art index data structure used in the database community, a separate chaining hash table (`hashtable`), a leaf-oriented unbalanced BST (`leaftree`), a leaf-oriented balanced BST (`leaftreap`) with an optimization that stores a batch of key-value pairs (up to 2 cachelines worth) in each leaf to minimize height, and an (a,b)-tree (`abtree`). To support concurrent accesses, the data structures use fine-grained, optimistic locking[6], as in [33, 83, 99, 104]. This approach involves (1) traversing the data structure without any locks, (2) locking a neighborhood around the nodes you wish to modify, (3) checking for consistency, and (4) performing the desired modifications. If the consistency check fails, locks are released and the operation restarts. Read-only operations do not take any locks.

We implement a tryLock and a strictLock version of each data structure. Both tryLock and strictLock can either be lock-free (with helping and logging) or blocking (using test-and-test-and-set locks), and with our library, this choice can be made by changing a flag at runtime.

To the best of our knowledge, this results in the first lock-free implementation of an adaptive radix tree. In many workloads, our lock-free `arttree` significantly outperforms the other lock-free ordered set data structures that we ran. Our implementations of these optimistic, fine-grained locking data structures are available at `https://github.com/cmuparlay/flock`.

3.6 Experimental Evaluation

Our experimental evaluation has two main goals: first, to compare the performance of lock-free locks with blocking locks and second, to compare data structures written with lock-free locks with state-of-the-art alternatives.

Setup. Our experiments ran on a 72-core Dell R930 with 4x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. The machine's interconnection layout is fully connected so all four sockets are equidistant from each other. We interleaved memory across sockets using `numactl -i all`. The machine runs Ubuntu 16.04.6 LTS. We compiled using g++ 9.2.1 with `-O3`. We used ParlayLib [31] for scalable memory allocation.

Workloads. We experiment with set data structures supporting `insert`, `delete` and `lookup` with 8-byte keys and 8-byte values. Our experiments follow a similar methodology to previous papers [8, 48]. We first pick a key range $[1, r]$ and prefill the data structure with half the keys in the range. Then each thread performs a mix of `lookup` and update operations, where update operations are evenly split between `inserts` and `deletes`, keeping the data structure size stable throughout the run. Each experiment is run for 3 seconds (sufficient for reaching a stable state) and repeated 4 times. The first run is a warmup run and an average of the last 3 runs is reported. Before the warmup run, we shuffle the ParlayLib memory allocator by allocating a large number

[6]Also known as optimistic synchronization [85].

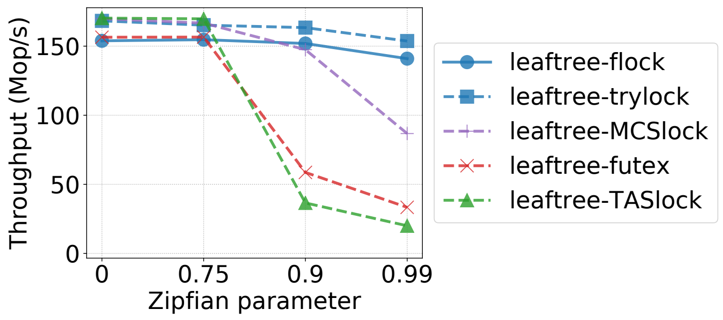


Figure 3.5: Comparing our lock-free implementation of locks with various blocking implementations on a lock-based binary search tree initialized with 100K keys. Workload consists of 144 threads, and 50% updates.

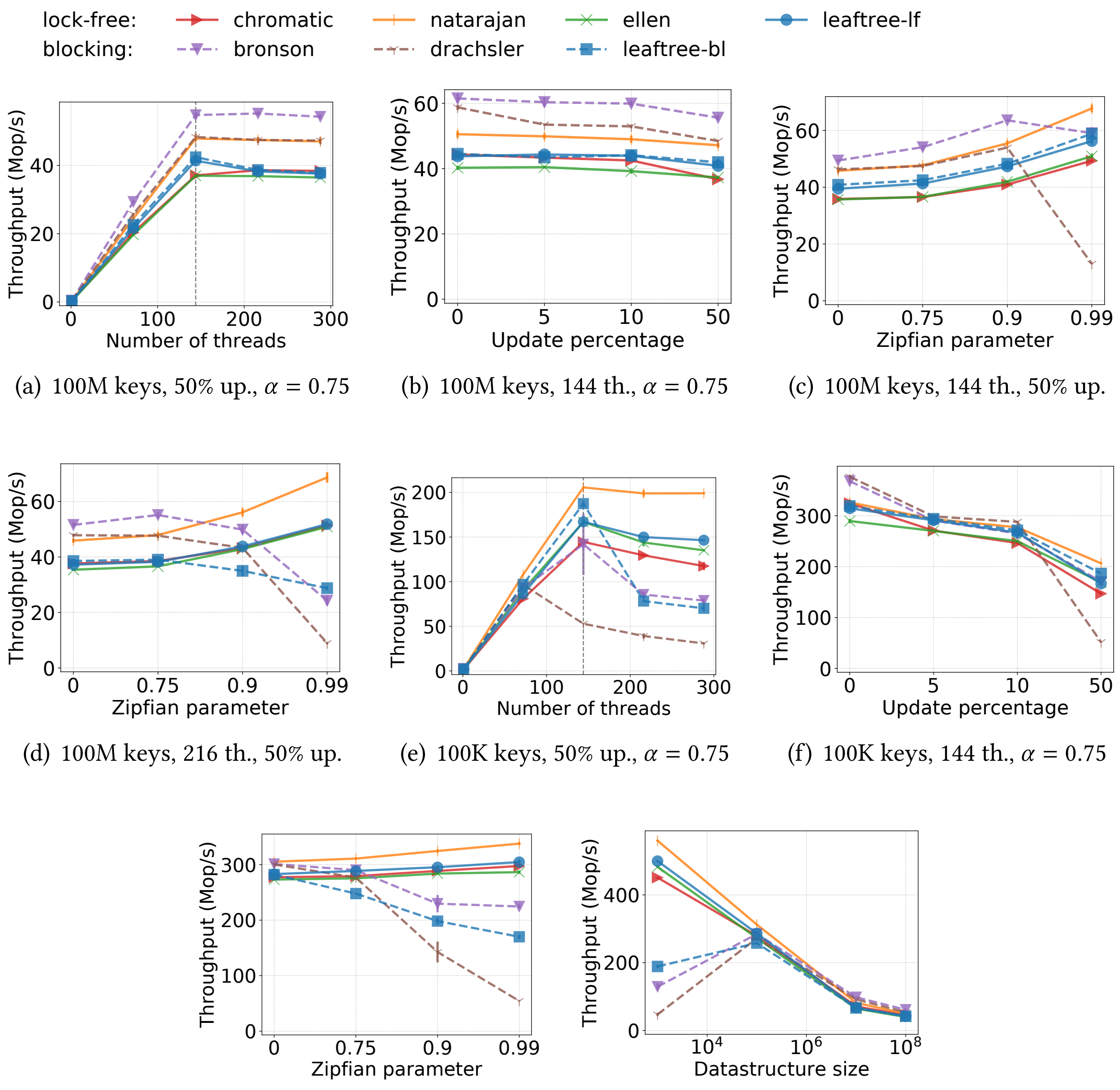
of nodes and freeing them in a random order to increase consistency across runs. Standard deviation between runs is small enough that the error bars in our graphs are only visible for a small number of data points.

All keys are randomly chosen from the range $[1, r]$ according to a zipfian distribution parameterized by α. Zipfian with $\alpha = 0$ is identical to the uniform distribution and higher α skews accesses towards certain "hot" keys, which is more representative of real-world workloads. The zipfian distribution is also used in the YCSB benchmark suite, which mimics OLTP index workloads [43]. We mostly run with 5% and 50% updates, following YCSB Workloads B and A, respectively.

Our experiments vary four parameters: data structure size, update rate, α, and number of threads. We show graphs along each of the dimensions, fixing the other three. Since `arttree` is a trie data structure, it benefits heavily from densely packed keys, so we sparsify the key range by hashing each key from $[1, r]$ to a 64-bit integer. This does not affect the other data structures since they either are purely comparison based or hash the keys themselves.

Try vs strict lock. In data structures that employ optimistic locking, tryLock is often preferable to strictLock. This is because optimistic locking requires checking for consistency after taking the necessary locks. So if a process p_1 tries to acquire a lock that is held by a another process p_2, it is better for p_1 to restart its operation instead of waiting to acquire the lock because it will likely fail its consistency check due to modifications by p_2. We see this happen in the `leaftree` in Figure 3.5, which compares various tryLock and strictLock implementations. The `leaftree-flock` algorithm uses our lock-free version of tryLock, whereas `leaftree-trylock` uses the traditional blocking version. For strictLocks, we tested `MCSlock`, a queue based lock by Mellor-Crummey and Scott [112], `futex`, a fast user space lock provided by linux [108], and also `TASlock`, a test-and-test-and-set based spinlock. We see that the higher the Zipfian parameter α is, the more contention there is on the locks, and the more beneficial tryLock becomes. This holds for both blocking locks and lock-free locks. In the rest of this section, we only report on tryLocks.

Binary trees. Figure 3.6 shows the throughput of concurrent trees under a wide range of workloads. We compare our tree implementations with state-of-the-art lock-based (Bronson [33], Drachsler [58]) and lock-free (Ellen [62], Chromatic [38] and Natarajan [122]) binary search



(a) 100M keys, 50% up., $\alpha = 0.75$

(b) 100M keys, 144 th., $\alpha = 0.75$

(c) 100M keys, 144 th., 50% up.

(d) 100M keys, 216 th., 50% up.

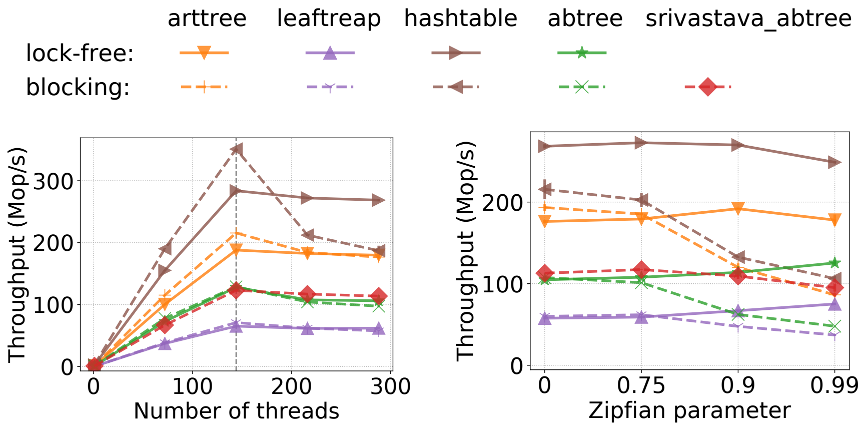
(e) 100K keys, 50% up., $\alpha = 0.75$

(f) 100K keys, 144 th., $\alpha = 0.75$

(g) 100K keys, 216 th., 5% up.

(h) 216 th., 5% up., $\alpha = 0.75$

Figure 3.6: Throughput of binary trees under a variety of workloads are shown. Dotted lines are used for blocking data structures and solid lines for lock-free ones. Subcaptions abbreviate 'threads' to 'th' and 'updates' to 'up'. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of our locks, respectively.



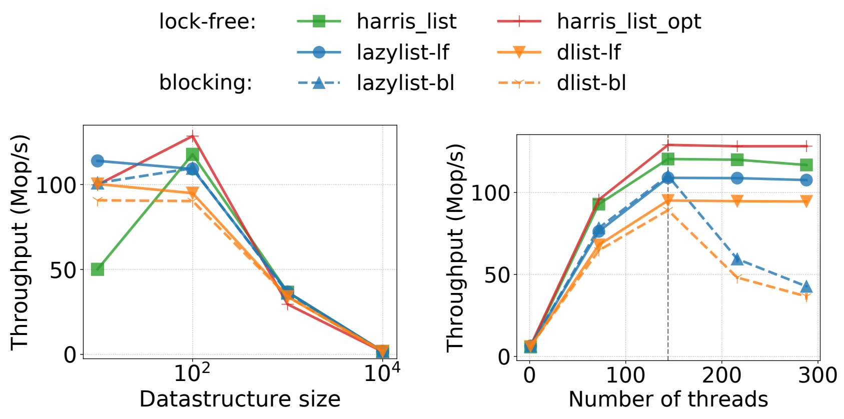
(a) 100M keys, 50% up., $\alpha = 0.75$ (b) 100M keys, 216 th., 50% up.

Figure 3.7: Throughput of concurrent set data structures.

trees. These implementations were obtained from the SetBench benchmarking suite [8]. Bronson and Chromatic are the only balanced tree among these implementations. Regarding the lock-free trees, Ellen and Natarajan are implemented directly from CAS whereas Chromatic is implemented using the higher-level LLX/SCX primitives [37]. Note that in all the graphs, lock-based algorithms are denoted by dotted lines and lock-free algorithms appear as solid lines.

Figures 3.6(a)- 3.6(d) consider the case where the tree does not fit in cache and Figures 3.6(e)-3.6(g) consider the case where they do. In out-of-cache workloads, performance is dominated by cache misses incurred during the traversal phase. Figure 3.6(b) shows that the cost of updating the tree is small compared to these cache misses, whereas in Figure 3.6(f), increasing the percentage of updates significantly reduced throughput. All trees scale well, up until oversubscription (Figures 3.6(a) and 3.6(e)), with the exception of Drachsler in Figure 3.6(e). Bronson is generally the fastest when tree size is large because it is better balanced compared to the other trees (many of which are only balanced in expectation due to random inserts), resulting in shorter traversals and less cache misses. As the zipfian parameter α increases, all trees except Bronson and Drachsler speed up because higher α means more locality and less cache misses (Figure 3.6(c)). However, large α also means more contention. In the case of Bronson and Drachsler, which both use blocking strict-locks, this extra contention out-weighs the benefits of locality. This effect is even more severe for small trees (Figure 3.6(g)).

Lock-free vs blocking. Next, we compare the throughput of lock-free data structures with blocking ones, with particular emphasis on `leaftree-lf` and `leaftree-bl`, the lock-free and blocking variants, respectively, of our `leaftree`. The overhead of lock-free locks come from two main sources (1) allocating and initializing a new descriptor every time a lock is acquired, and (2) committing values to the log during critical sections. A successful `insert` commits about 5 entries to the log. This overhead is only visible in small trees with high update rates (Figures 3.5 and 3.6(e)). Across all the graphs in Figure 3.6, the overhead of using lock-free locks rather than traditional blocking locks is no more than 11%. Furthermore, most graphs do not show any visible overhead.



(a) 144 th., 5% up., $\alpha = 0.75$ (b) 100 keys, 5% up., $\alpha = 0.75$

Figure 3.8: Throughput of singly and doubly linked lists. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of our locks, respectively.

Where lock-free algorithms shine is in oversubscribed cases (e.g. 288 threads) with high contention. This is because a thread may get descheduled while it is partway through an update, and in a lock-free algorithm, if another thread wants to update the same location, it can simply help complete the inactive thread's update and then proceed with its own. However, in a blocking data structure, the new thread will have to either wait for the inactive thread to be scheduled again and release its lock, or yield and context switch, both of which are expensive. This effect can be seen in the right side of Figures 3.6(d) and 3.6(g) and the left side of Figure 3.6(h) where the four lock-free trees outperform the three blocking trees. In particular, `leaftree-lf` outperforms `leaftreap-bl` by up to 2.4x in Figure 3.6(h).

Other set datatypes. In Figure 3.7, `arttree`, `leaftreap`, `abtree` and `hashtable`, generally follow the same pattern as `leaftree`. That is, lock-free versions outperform their blocking counterparts in oversubscribed, high contention scenarios (right side of Figure 3.7(b)), by up to 2.5x in the case of the `hashtable` and 2x for the `arttree`. In non-oversubscribed scenarios (left size of Figure 3.7(a)), the overhead of using lock-free locks is small, especially for `abtree` and `leaftreap`. The overhead of lock-free locks is highest in the `hashtable` because its search time (i.e. fraction of time spent outside of the critical section) is small and hence the overhead for the locked part plays a larger role. Figure 3.7 also compares our data structures with Srivastava's CoPub-ABtree [151], a state-of-the-art blocking (a,b)-tree. Our lock-free `abtree` performs similarly to `srivastava_abtree` in most cases but is up to 32% faster at the right of Figure 3.7(b).

Linked List Experiments. Figure 3.8 compares doubly and singly linked lists written using our lock-free locks (`dlist` and `lazylist`, respectively) with Harris's lock-free singly linked list [80] (`harris_list`), and an optimized version of Harris's list where `find` operations do not perform any helping [48] (`harris_list_opt`). In most cases, our lock-free `lazylist` is slower than `harris_list_opt` by about 16% because the descriptors in `harris_list_opt` are optimized to simply be flags. Interestingly, the lock-free versions of `dlist` and `lazylist` outperform their corresponding blocking versions even without oversubscription on small lists (left of Figure 3.8(a)).

The pseudo-code for `dlist` was presented earlier in Algorithm 3.1 and Figure 3.8 show that this simple algorithm performs well. The overhead of maintaining back pointers is only about 13% (comparing `dlist` with `lazylist`).

3.7 Conclusion

We presented a mechanism for implementing lock-free locks, and a library-based implementation. It is the first such library implementation of lock-free locks we know of. The approach is practical in two senses. Firstly, in terms of performance it is competitive with state-of-the-art lock-free and lock-based data structures. Secondly, using the library requires very few changes to existing lock-based implementations—basically wrapping shared values in a `flck::atomic`, and using the Flock lock structure and memory management. In terms of functionality it significantly extends previous suggested approaches to lock-free locks, supporting memory management, races, and tryLocks.

We separate out the idea of idempotent blocks of code (thunks) and present a general and efficient approach along with a C++-based library to support them. The approach supports arbitrary code with load, stores and CAMs on shared locations, as well as memory allocation and retirement from a shared pool. A thunk using the approach can be run any number of times with instructions interleaved in any way while behaving like it ran once. The approach uses a shared log for each thunk so that separate runs of the thunk see the same result. The idempotent construction could be of independent interest.

We implemented several data structures using the approach. With regards to the opening question of whether to be lock-free or not to be, the experiments clearly indicate the advantage of lock-freedom when processors are oversubscribed. Our experiments are some of the first on concurrent data structures that study this effect. The experiments also show that the overhead of being lock-free for our structures is relatively small (rarely more than 10%) and often hardly noticeable.

Related Work

As mentioned, the idea of lock-free locks was introduced by Turek, Shasha and Prakash [159] and Barnes [14]. The idea of helping dates back earlier, at least to Herlihy's work on wait-free simulations [84]. Many wait-free and lock-free algorithms achieve their progress guarantees by allowing processes to safely *help* each other complete their operations, although in quite specific ways instead of using a general mechanism. Help used for wait-free progress was formally studied by Censor-Hillel, Petrank and Timnat [41].

The idea of *idempotence* has been used in the literature a variety of contexts [18, 30, 30, 34, 49, 50, 95, 109]. Kruijf, Sankaralingam and Jha [50] give a nice overview although only up to 2012. More recent work has focused on using idempotence for fault tolerance (e.g., [18, 30, 109]). All these approaches rely on some form of "context saving". Idempotence has also been considered and characterized in the literature under different names. Timnat and Petrank [156] define a similar notion known as *parallelizable* code, which intuitively allows several processes to execute it without changing its effects.

In recent work, Ben-David and Blelloch in [17] use a randomized implementation of lock-free locks to show that when point contention on locks is constant, then operations can be completed in constant expected time. We use their definition of idempotence in this thesis. However, their focus is on theoretical efficiency and fairness guarantees of acquiring the locks, whereas in this thesis we focus on the practicality of the approach. As with previous approaches to idempotence, their approach relies on context saving.

Approaches for achieving idempotence and lock-freedom sit on a spectrum of generality. The focus of this thesis is to improve the practicality of the far side of the spectrum; fully general idempotence/lock-free constructions. However, many other approaches exist, which are less general but can be more efficient for their specific applications. For example, on the other end of the spectrum are hand-designed lock-free data structures. These data structures are often designed to be able to have 'critical sections' that contain just one CAS instruction, and can therefore be executed atomically in hardware with no locks. For example, Michael and Scott's queue [118] allows new nodes to be enqueued by swinging a single pointer. Idempotent help is given by later updating the tail pointer. Similar algorithms, like Harris's linked-list [80] and Natarajan and Mital's BST [122], make use of descriptors to allow others to help, but these descriptors are optimized to simply be flags. These approaches yield very fast lock-free data structures, but are difficult to generalize.

A middle-ground between generality and efficiency is found with approaches that implement useful primitives for lock-freedom. For example, Brown et al [37] introduce the LLX/SCX primitive, which allows atomically checking that several locations have not changed their values, ‘freezing’ some of them, and modifying one of them. This primitive can be seen as a lock with a restricted critical section. Another example of such a primitive is multi-word CAS, which allows several memory locations to be CASed atomically [68, 78, 81].

Some work aims at achieving practical lock-free locks but only partially solve the problem. Rajwar and Goodman describe a hardware-based technique that are lock-free under an assumption that processes do not fail or stall during certain critical regions [140]. We assume a process can fail or stall at any instruction. Gidenstam and Papatriantafilou [74] look at how to make the handoff of locks lock-free (i.e., waking up threads suspended on a lock in a lock-free manner), but a thread blocked during a lock will still delay any waiting threads indefinitely.

Part II

Linearizable Snapshots

Introduction

The ability to take a snapshot, i.e. obtain a consistent view, of shared memory is very useful for concurrent programming. For example, it can be used to query a data structure atomically across multiple locations, even as the data structure is being modified concurrently by other processes. Such *multi-point* queries have traditionally been notoriously difficult to implement efficiently. In this part, we present general techniques for taking snapshots of CAS objects and pointer types. We show that multi-point queries implemented using these general techniques are often more efficient than data structures specifically designed to support those kinds of queries.

Chapter 4

Versioned CAS

4.1 Introduction

Given any concurrent data structure, this chapter presents an approach for efficiently taking snapshots of its constituent CAS objects. Importantly, this approach preserves all the time bounds and parallelism of the original algorithm/data structure. The interface is based on creating a *camera* object that has a collection of associated *versioned CAS objects*, which support read and CAS operations like normal CAS objects, as well as a versioned read operation. The camera object supports a single operation `takeSnapshot` that takes a snapshot of the values stored in all the associated versioned CAS objects. The `takeSnapshot` operation does not make a copy of these objects. Instead it returns, in constant time, a handle representing the point in time the snapshot was taken. In our algorithms, this handle is a timestamp. The handle can later be used to query (via the versioned read operation) the state of any versioned CAS object at that time. New versioned CAS objects can be associated with an existing camera object, so our construction is applicable to dynamically-sized data structures.

This interface is more flexible than the one traditionally used for a *snapshot object* [2], which stores an array and provides *update* operations that write to individual components and *scan* operations that return the state of the entire array. Instead of creating a copy of the state of the entire shared memory in the local memory of a process, our `takeSnapshot` simply makes it possible for a process to later read *only* the memory locations it needs from shared memory, knowing that the collection of all such reads will be atomic. Although partial snapshot objects [10, 89] allow scans of part of the array, they require the set of locations to be specified in advance, whereas our approach allows the locations to be chosen dynamically as a query is executed.

Our algorithm has the following important properties.

1. Taking a snapshot of the current state and returning a handle to it takes a constant number of instructions.
2. A CAS or read of the current state of a versioned CAS object takes constant time. Therefore, adding snapshots to a CAS-based data structure preserves the data structure's asymptotic time bounds.

3. Reading the value of a versioned CAS object from a snapshot takes time proportional to the number of successful CAS operations on the object after the snapshot and before the start of the read. Thus, all reads are *wait-free* (i.e., every read is completed within a finite number of instructions.)
4. The algorithm is implemented using single-word read and CAS, which are supported by modern architectures. It uses an unbounded counter.

We know of no previous general mechanism for snapshotting the state of memory satisfying even the first two properties.

Similarly to previous work [25, 69, 98, 124, 144, 147, 171], we use a version list for each CAS object. The list has one node per update (successful CAS) on the object. Each node contains the value stored by the update and a *timestamp* indicating when the update occurred. The list is ordered by timestamps, most recent first. A difficulty in implementing version lists without locks, which we address, is the need to add a node to the version list, read a global timestamp, and save that timestamp in the node, all atomically.

Snapshots and Multi-point Queries. Our interface provides a simple way of converting a concurrent data structure built out of CAS objects into one that supports snapshots: simply replace all CAS objects with our versioned CAS objects, all associated with a single camera object. If all shared mutable state is stored in the CAS objects, then taking a snapshot will effectively provide access to an atomic copy of the entire state of the data structure at the snapshot's linearization point. After taking a snapshot, a read-only query is free to visit any part of the data structure state at its leisure, even as updates proceed concurrently. Often, the query can be performed by simply taking a snapshot and then running a standard sequential algorithm on the data structure by replacing each read with our versioned read.

In Section 4.4, we define more precisely when multi-point queries can be computed from snapshots. In particular, we discuss how our approach can be used for arbitrary queries on Michael-Scott queues [118], Harris's linked-lists [80], and two different binary search trees [38, 62]. On the binary search trees, for example, one can support atomic queries for finding the smallest key that matches a condition, reporting all keys in a range, determining the height of the tree, or searching for a set of keys. The time complexity of each query is bounded above by the sequential cost of the query plus the number of vCAS operations it is concurrent with.

Avoiding Indirection and Other Optimizations. Our algorithm introduces only constant overhead for existing operations, and allows the implementation of wait-free queries. However, our construction does introduce a level of indirection: to access the value of a versioned CAS object, one must first access a pointer to the head of the version list, which leads to the actual value. This may introduce an extra cache miss per access. We therefore consider an optimization to avoid this in Section 4.5. This optimization applies to many concurrent data structures that satisfy the *recorded-once* property we introduce. Roughly speaking, recorded-once means that each data structure node is the new value of a successful CAS at most once. This allows us to store information for maintaining the version lists (in particular the timestamp and the pointer to the next older version) directly in the nodes themselves, thus removing the level of indirection (see Figure 4.2 for an example). In Section 4.6, we describe other optimizations to reduce contention and shorten version lists.

Memory Reclamation. Maintaining all old versions of a versioned CAS object may be infeasible. In Section 4.7, we describe how to garbage collect old versions using an approach based on Epoch Based Memory Reclamation (EBR) [71]. Experiments indicate that our approach works well in practice and has low memory overhead.

Implementation and Experiments. To study the time and space overhead of our approach, we applied it to three existing concurrent binary search trees, one balanced and two not [7, 38, 62]. Adding support for snapshots was very easy and required minimal changes to the original code. The experiments demonstrate that the overhead is small. For example, the time overhead of supporting snapshots is about 9% for a mix of updates and queries on the current version of the tree. We also compare to state-of-the-art data structures that support atomic range queries, including `KiWi` [15], `LFCA` [170], `PNB-BST` [67], and `SnapTree` [33]. In almost all cases, our data structure performs as well as or better than all of these special-purpose structures even though our approach is general purpose. Finally, we implement a variety of other atomic multi-point queries and show that the overhead compared to non-atomic implementations, which are correct only when there are no concurrent updates, is small.

Contributions. In summary, our contributions are:

- A simple, constant-time approach to take a snapshot of a collection of CAS objects.
- A technique to use snapshots to implement linearizable multi-point queries on many lock-free data structures.
- Optimizations that make the technique more practical, for example, by avoiding indirection.
- Experiments showing our technique has low overhead, often outperforming other state-of-the-art approaches, despite being more general.

4.2 Versioned CAS Objects

Our approach uses "time-stamped" versioned lists to maintain the state of each object, as in previous work (e.g., [25, 69, 98, 124, 144, 171]). Unlike most of this work, updates do not increment the timestamps—only taking a snapshot might increment the timestamp.[1] An important aspect of our algorithm is how it attaches a timestamp to a new version when updating an object (with a CAS). This involves temporarily setting the new version's timestamp to an undetermined value (TBD) and then updating this to the "current" timestamp only after it is inserted into a version list. The new version's timestamp might be updated by the CAS that created it or via helping by a concurrent operation accessing the object. We refer to this idea as *set-stamp helping* and the helping step is crucial.

We begin with a sequential specification of our objects.

Definition 2 (Camera and Versioned CAS Objects)**.** *A* versioned CAS *object stores a* value *and supports three operations,* `vRead`, `vCAS`, *and* `readVersion`. *The first two operate on the* current value *and the third is used to access a value captured by a snapshot. A* camera *object supports a*

[1]When there are concurrent snapshots, only one needs to increment the timestamp, avoiding sequentializing snapshots.

single operation, `takeSnapshot`*. Each versioned CAS object O is associated with a single camera object when it is created. Consider a sequential history of operations on a camera object S and the set Λ_S of* `vCAS` *objects associated with it. The behavior of operations on S and O, for all $O \in \Lambda_S$, is specified as follows:*

- *An O.*`vCAS(oldV, newV`*) attempts to update the value of O to* `newV` *and this update takes place if and only if the current value of O is* `oldV`*. If the update is performed, the* `vCAS` *operation returns* `true` *and is* successful. *Otherwise, the* `vCAS` *returns* `false` *and is* unsuccessful.
- *An O.*`vRead`*() returns the current value of O.*
- *The behavior of* `readVersion` *and* `takeSnapshot` *are specified simultaneously. A precondition of O.*`readVersion`*(ts) is that there must have been an earlier S.*`takeSnapshot`*() that returned the handle ts. For any S.*`takeSnapshot`*() operation T that returns ts and any O.*`readVersion`*(ts) operation R, R must return the value O had when T occurred.*

4.2.1 A Linearizable Implementation

Algorithm 4.1 is a linearizable implementation of versioned CAS and camera objects. The ideas behind the algorithm are described below.

```
class Camera {
  int timestamp;
  Camera() { timestamp = 0; }
  int takeSnapshot() {
    int ts = timestamp;
    CAS(&timestamp, ts, ts+1);
    return ts; }
};
class VNode {
  Value val; VNode* nextv; int ts;
  VNode(Value v, VNode* n){
    val = v; ts = TBD; nextv = n;}
};
class VersionedCAS {
  VNode* VHead;
  Camera* S;
  VersionedCAS(Value v, Camera* s){
    S = s;
    VHead = new VNode(v, NULL);
    initTS(VHead); }
  void initTS(VNode* n) {
    if(n->ts == TBD) {
      int curTS = S->timestamp;
      CAS(&(n->ts), TBD, curTS); }}
```

```
// class VersionedCAS continued...
Value readVersion(int ts) {
  VNode* node = VHead;
  initTS(node);
  while (node->ts > ts)
    node = node->nextv;
  return node->val; }
Value vRead() {
  VNode* head = VHead;
  initTS(head);
  return head->val; }
bool vCAS(Value oldV, Value newV) {
  VNode* head = VHead;
  initTS(head);
  if(head->val != oldV) return false;
  if(newV == oldV) return true;
  VNode* newN = new VNode(newV, head);
  if(CAS(&VHead, head, newN)) {
    initTS(newN);
    return true;
  } else {
    delete newN;
    initTS(VHead);
    return false; } } };
```

Algorithm 4.1: Linearizable implementation of a camera object and a versioned CAS object.

The Camera Object. The camera object behaves like a global clock for all versioned CAS objects associated with it. It is implemented as a counter called `timestamp` that stores an integer value.

A takeSnapshot simply returns the current value *ts* of variable timestamp as the handle and attempts to increment timestamp using a CAS. If this CAS fails, it means that another concurrent takeSnapshot has incremented the counter, so there is no need to try again. The handle will be used by future readVersion operations to find the latest version of any versioned CAS object that existed when the counter was incremented from *ts* to $ts + 1$.

The Versioned CAS Object. Each versioned CAS object is implemented as a singly-linked list (a *version list*) that preserves all earlier values committed by vCAS operations, where each version is labeled by a timestamp read from the camera's counter during the vCAS. The list is ordered with more recent versions closer to the head of the list. A vRead operation just returns the version at the head of the list. A successful vCAS adds a node to the head of the list. *After* the node has been added to the list, the value of the camera object's counter is recorded as the node's timestamp. A readVersion(*ts*) operation traverses the version list and returns the value in the first node with timestamp at most *ts*.

The versioned CAS object stores a pointer VHead to the last node added to the object's version list. Each node in this list is of type VNode and stores

- a value val, which is immutable once initialized,
- a timestamp ts, and
- a pointer nextv to the next VNode of the list, which contains the next (older) version of the object.

The version list essentially stores the history of the object.

Timestamps. We use a special timestamp TBD (to-be-decided) as the default timestamp for any newly-created VNode. TBD is not a valid timestamp and must be substituted by a concrete value later, once the VNode has been added to the version list. When a VNode x is added to the version list, we call the initTS subroutine (Lines 22–24) to assign it a valid timestamp read from the camera object's timestamp field. Once x's timestamp changes from TBD to a valid value, it will never change again, because the CAS on Line 24 succeeds only if the current value is TBD. This initTS function can be performed either by the process that added x to the list, or by another process that is trying to help.

Implementing readVersion(ts) and vRead(). The goal of a readVersion(ts) is to return the latest version whose timestamp is at most ts. It first reads VHead and, if necessary, helps set the timestamp of the VNode that VHead points to by calling initTS. The readVersion then traverses the version list by following nextv pointers until it finds a version with timestamp smaller than or equal to ts, and returns the value in this VNode. The vRead function looks only at VHead, helps set the timestamp of the VNode that VHead points to, and returns the value in that VNode.

Implementing vCAS(oldV, newV). This operation first reads VHead into a local variable head. Then it calls initTS on head to ensure its timestamp is valid. If the value in the VNode that head points to is not oldV, the vCAS operation fails and returns false (Line 39). Otherwise, if oldV equals newV, the vCAS returns true because nothing needs to be updated. This is not just an optimization that avoids creating another VNode unnecessarily; it is also required for correctness because without it, a successful vCAS(a, a) could cause a concurrent vCAS(a, b) to fail. If oldV and newV are different, and the VNode that head points to contains the value oldV, the algorithm attempts to add a new VNode with value newV to the version list. It first allocates a new VNode newN (Line 41) to store newV and lets it point to head as its next version. It then attempts to add newN to the beginning of the list by swinging the pointer VHead from head to newN using a CAS (Line 42). If successful, it then calls initTS on the new VNode to ensure its timestamp is valid, and returns true to indicate success. Before this call to initTS terminates, a valid timestamp will have been recorded in the new VNode, either by this initTS or by another operation helping the vCAS.

If the CAS on Line 42 fails, then VHead must have changed during the vCAS, and this change must have been done by a concurrent successful vCAS. In this case, the new VNode is not appended to the version list. The algorithm deallocates the new VNode (Line 46) and returns false. The unsuccessful vCAS also helps the first VNode in the version list acquire a valid timestamp.

Helping. As mentioned, a vRead, readVersion and an unsuccessful vCAS all help (by calling initTS) to ensure that the timestamp of the VNode at the head of the version list is valid before they return. This is necessary to overcome the main difficulty in implementing version lists without locks, i.e., making the following three steps appear atomic: adding a node to the version list, reading a global timestamp, and recording a valid timestamp in the node. (See the discussion of correctness below.)

Initialization. We assume that the constructor (Line 3) for the camera object completes before invoking the constructor (Line 17) for any associated versioned CAS object. We require, as a precondition of any readVersion(*ts*) operation on a versioned CAS object O, that O was created before the takeSnapshot operation that returned the handle *ts* was invoked. In other words, one should not try to read the version of O in a snapshot that was taken before O existed. When using versioned CAS objects to implement a pointer-based data structure (like a tree or linked list), this constraint will be satisfied naturally.

Correctness. Theorem 4.2.1 states the algorithm's properties.

Theorem 4.2.1 (Linearizability and Time Bounds)**.** *Algorithm 4.1 is a linearizable implementation of versioned CAS and camera objects such that*

1. *the number of instructions performed by* vRead*,* vCAS*, and* takeSnapshot *is constant, and*
2. *the number of instructions performed by the operation O.*readVersion*(ts) is proportional to the number of successful O.*vCAS *operations linearized between the linearization point of the* takeSnapshot *operation that returned ts and the start of the* readVersion*.*

A complete proof of Theorem 4.2.1 appears in the next Section. Below, we just describe the linearization points used in that proof. We say that a timestamp of a VNode is *valid* at some point if the ts field is not TBD at that point; it is *invalid* otherwise.

- For a vCAS operation V:
    - If V performs a successful CAS on Line 42 adding a node x to the version list, and x's timestamp eventually becomes valid, then V is linearized on Line 23 of the initTS method that makes x's timestamp valid.

    - Let x be the node `VHead` points to on Line 37 of V. If V returns on Line 39 or 40, it is linearized either at Line 37 if x's timestamp is valid at that time, or the first step afterwards that makes x's timestamp valid.
    - If V returns `false` on Line 48, then V failed its `CAS` on Line 42. Thus, some other `vCAS` operation changed `VHead` after V read it at Line 37. We linearize the `vCAS` immediately after the linearization point of the `vCAS` operation V' that made the *first* such change. If several `vCAS` operations that return on Line 48 are linearized immediately after V', they can be ordered arbitrarily.
- For a `vRead` operation that terminates, let x be the `VNode` read from `VHead` at Line 33. The `vRead` is linearized at Line 33 if x's timestamp is valid at that time, or at the first step afterwards that makes x's timestamp valid.
- A `readVersion` operation that terminates is linearized at its last step.
- For a `takeSnapshot` operation T that terminates, let ts be the value read from `timestamp` on Line 5, T is linearized when `timestamp` changes from ts to $ts + 1$. This increment could have either been performed by T or a `takeSnapshot` concurrent with T.

Intuitively, the correctness of an O.`readVersion` operation depends on ensuring that the timestamp associated with a value is *current* (i.e., in the `timestamp` field of the camera object S associated with O) at the linearization point of the `vCAS` that stored the value in O. Hence, we linearize a successful `vCAS` at the time when the successfully installed timestamp was read from S. Thus, a `VNode` x can appear at the head of the version list before the `vCAS` that created x is linearized. This is why any other operation that finds a `VNode` with an invalid timestamp at the head of the version list calls `initTS` to help install a valid timestamp in it before proceeding. This helping mechanism is crucial to prove that the linearization points described above are well-defined and within the intervals of their respective operations.

4.3 Proof of Linearizability

In this section, we prove that Fig. 4.1 is a linearizable implementation of versioned CAS and camera objects. First we argue that it suffices to prove linearizability for histories consisting of a single versioned CAS object and a single camera object. Suppose two versioned CAS objects are associated with different camera objects. Then we can prove linearizability for the two sets of objects independently because they do not access any common variables and do not affect each other in terms of sequential specifications. Suppose two versioned CAS objects O_1 and O_2 are associated with the same camera object S. Let H' be an execution of operations on these three objects. Furthermore, let H'_1 be the execution H' restricted to only operations from S and O_1, and similarly, let H'_2 be the execution H' restricted to only operations from S and O_2. We will define the linearization points of S so that they are not affected by operations on O_1 or O_2. Therefore, showing that both H'_1 and H'_2 are linearizable is sufficient for showing that H' is linearizable because S will be linearized the same way in both H'_1 and H'_2.

Let H be an execution of a versioned CAS object O and a camera object S. We assume that S and O are initialized by their constructors (Line 3 and 17, respectively) before the beginning of H. We assume this execution satisfies the precondition (described in Definition 2) that whenever readVersion(*ts*) is invoked, there must be a completed takeSnapshot operation that returned *ts*. When referring to the variables *O*.VHead and *S*.timestamp, we will often abbreviate them to VHead and timestamp.

We first review some useful terminology. Recall that a VNode has a *valid* timestamp at some configuration *C* if the value of its ts field is not TBD at *C*. Otherwise, the timestamp of the node is called *invalid*. We use the term *version list* to refer to the list that results from starting at the VNode pointed to by VHead and following nextv pointers. The *head* of the version list is the VNode pointed to by VHead.

A *modifying* vCAS operation is one that performs a successful CAS on line 42. Due to the if statement on line 40, if vCAS($oldV$, $newV$) is a modifying vCAS operation, then $oldV \neq newV$. Note that modifying vCAS operations can return only on line 44 and any operation that returns on line 44 is a modifying vCAS. A vCAS is *successful* if it is a modifying vCAS or if it returns true at line 40. Otherwise, it is *unsuccessful*.

We first show that the only change to a version list is inserting a VNode at the beginning of it.

Lemma 4.3.1. *Once a* VNode *is in the version list, it remains in the version list forever.*

Proof. The only way to change a version list is a successful CAS at line 42, which changes VHead from head to newN. When this happens, newN->nextv = head, so all VNodes that were in the version list before the CAS are still in the version list after the CAS. □

It is easy to check that every time we access some field of an object via a pointer to that object, the pointer is not NULL. VHead always points to a VNode after it is initialized on Line 19 of *O*'s constructor. It follows that every call to initTS is on a non-null pointer. The precondition of readVersion(ts) ensures that ts is a timestamp obtained from *S* after *O* was initialized and is therefore greater than or equal to the timestamp that *O*'s constructor stored in the initial VNode of the version list. Thus, the readVersion will stop traversing the version list when it reaches that initial VNode, ensuring that node is never set to NULL on line 30.

Linearization Points. We begin with a few simple lemmas that describe when VNodes have valid timestamps.

Lemma 4.3.2. *The following hold:*

1. *Before* initTS *is called on a* VNode, VHead *has contained a pointer to that* VNode.
2. *After a complete execution of* initTS *on some* VNode*, that* VNode*'s timestamp is valid.*

Proof. All calls to initTS are done on a pointer that has either been read from VHead or successfully CASed into VHead. Once a timestamp is valid, it can never be modified again, since only a CAS on line 24 modifies the ts variable of any VNode. The CAS on Line 24 can fail only if the ts variable is already a valid timestamp. □

Lemma 4.3.3. *In every configuration C, the only* VNode *in the version list that can have an invalid timestamp is the head of the version list.*

Proof. No VNode's nextv pointer changes after the VNode is created, so the only way the version list can change is when VHead is updated. Moreover, no VNode's timestamp ever changes from valid to invalid. So, we must only show that updates to VHead preserve the claim.

The value of VHead changes only when a successful CAS is executed on Line 42 of an instance of vCAS. Consider any such successful CAS by some process p and assume the claim holds in the configuration before the CAS to show that it holds immmediately after the CAS. This CAS changes VHead from head to newN. By the initialization of newN on Line 41, that VNode's nextv pointer is head. So, we must show that head and all VNodes reachable from head by following nextv pointers have valid timestamps when the CAS occurs. Before executing this CAS, p executes initTS(head), so, by Lemma 4.3.2(2), that VNode's timestamp is valid at the time that the CAS is executed. Since the CAS is successful, VHead was equal to head immediately before the CAS, so all nodes reachable from that VNode had valid timestamps, by our assumption. □

The next lemma is used to define the linearization point of a modifying vCAS.

Lemma 4.3.4. *Suppose an invocation of* initTS *makes the timestamp of some* VNode *n valid. Then, n is the head of the version list when that* initTS *executes Line 23 and 24.*

Proof. By Lemma 4.3.2(1), every call to initTS is on a pointer that has previously been in VHead, so n has been in the version list before initTS is called. By Lemma 4.3.1, n is still in the version list when Line 23 and 24 are executed. By Lemma 4.3.3, n remains at the head of the version list until its timestamp becomes valid when initTS performs Line 24. □

We are now ready to define linearization points. As we define them, we argue that the linearization point of each operation is well-defined and within the interval of the operation.

- A vCAS operation is linearized depending on how it executes.
    - If the vCAS performs a successful CAS on Line 42 that adds a node n to the version list, and n's timestamp eventually becomes valid, then the vCAS is linearized on Line 23 of the initTS method that makes n's timestamp valid. Lemma 4.3.4 implies that the linearization point occurs after the vCAS adds n to the version list at Line 42. If the vCAS terminates, it first calls initTS on n at line 43, so Lemma 4.3.2(2) ensures the vCAS is linearized and that the linearization point comes before the end of that initTS.
    - Let h be the value of VHead at Line 37 of a vCAS operation. If the vCAS operation returns on Line 39 or 40, then it is linearized either at Line 37 if h's timestamp is valid at that time, or the first step afterwards that makes h's timestamp valid. Lemma 4.3.2(2) ensures this step exists and is within the interval of the vCAS, since initTS is called on h at line 38.
    - Finally, consider a vCAS(oldV, newV) operation V that returns false on Line 48. This is the most subtle case. The return on Line 48 is only reached when V fails its CAS on Line 42 because some other vCAS operation changed VHead after V read it at Line 37. We linearize the vCAS immediately after the vCAS operation V' that made the *first* such change. (If several vCAS operations that return on Line 48 are linearized immediately after V', they can be ordered arbitrarily.)

To argue that this linearization point is well-defined, we must show that the VNode n that V' added to the version list gets a valid timestamp, so that V' is assigned a linearization point as described in the first paragraph above. By Lemma 4.3.1, n is still in the version list when V reads VHead at Line 47. If n is no longer at the head of the version list, then n's timestamp must be valid, by Lemma 4.3.3. Otherwise, if n is still the head of the version list, then n's timestamp is guaranteed to be valid after V calls initTS on n (Line 47), by Lemma 4.3.2(2). So, in either case, V' is assigned a linearization point, which is before the timestamp of n becomes valid. Thus, V' (and therefore V) is linearized before the end of V. Lemma 4.3.4 implies that the linearization point of V' (and therefore of V) is after V' adds n to the version list, which is after V reads VHead. This proves that V's linearization point is inside the interval of V.

- For a vRead operation that terminates, let h be the VNode read from VHead at Line 33. The vRead is linearized at Line 33 if h's timestamp is valid at that time, or at the first step afterwards that makes h's timestamp valid. Lemma 4.3.2(2) ensures that this step exists and is during the interval of the vRead, since the vRead calls initTS on h at Line 34.
- A readVersion operation that terminates is linearized at its last step.
- For takeSnapshot operations, let t be the value read from timestamp on line 5. A takeSnapshot operation that terminates is linearized when the value of timestamp changes from t to $t + 1$. We know that this occurs between the execution of Line 5 and 6: either the takeSnapshot operation made this change itself if the CAS at line 6 succeeds, or some other takeSnapshot operation did so, causing the CAS on line 6 to fail.

Note that all operations that terminate are assigned linearization points. In addition, some vCAS operations that do not terminate are assigned linearization points.

Proof that Linearization Points are Consistent with Responses. Recall that H is the execution that we are trying to linearize. In the rest of this section, we prove that each operation returns the same response in H as it would if the operations were performed sequentially in the order of their linearization points.

Lemma 4.3.5. *Assume* VHead *points to a node h in some configuration C. If h.*ts *is valid in C then either h is the* VNode *created by the constructor of O, or the* vCAS *that created h is linearized before the configuration that immediately precedes C.*

Proof. Suppose h.ts is valid in C but h is not the VNode created by the constructor of O. Then h is created by some vCAS operation V that added h to the head of the version list. Since h.ts is valid in C, some step prior to C set h.ts by executing Line 24. The linearization point of V is at the preceding execution of Line 23. Thus, the linearization point precedes the configuration before C. □

We define the *value of the versioned CAS object in configuration C* to be the value that a versioned CAS object would store if all of the vCAS operations linearized before C are done sequentially in linearization order (starting from the initial value of the versioned CAS object).

The following crucial lemma describes how the value of the versioned CAS object is represented in our implementation. It also says that the responses returned by all `readVersion` and `vCAS` operations are consistent with the linearization points we have chosen.

Lemma 4.3.6. *In every configuration C of H after the constructor of the versioned CAS object has completed,*

1. *if* `VHead` *points to the* `VNode` *created by the constructor of the versioned CAS object, then* `VHead->val` *is the value of the versioned CAS object,*
2. *if the linearization point of the* `vCAS` *that created the first node in the version list is before C, then* `VHead->val` *is the value of the versioned CAS object, and*
3. *otherwise,* `VHead->nextv->val` *is the value of the versioned CAS object.*

Moreover, each `vRead` *and* `vCAS` *operation that is linearized at or before C returns the same result in H as it would return when all operations are performed sequentially in their linearization order.*

Proof. We prove this by induction on the length of the prefix of H that leads to C. In the configuration immediately after the constructor of the versioned CAS object terminates, `VHead->val` stores the initial value of the versioned CAS object.

Since `nextv` and `val` fields of a `VNode` do not change after the `VNode` is created, we must only check that the invariant is preserved by steps that modify `VHead` or are linearization points of `vCAS` operations (which may change the value of the versioned CAS object) or `vRead` operations. We consider each such step s in turn and show that, assuming the claim holds for the configuration C before s, then it also holds for the configuration C' after s.

First, suppose s is a successful CAS on `VHead` at line 42 of a `vCAS` operation. It changes `VHead` from `head` to `newN`, where `newN->next = head`. By Lemma 4.3.3, `head->ts` is valid when this CAS occurs, since `head` becomes the second node in the version list. By our assumption, the value of the versioned CAS object prior to the CAS is `head->val`. Since this step is not the linearization point of any `vCAS` operation, the value after the CAS is still `head->val`. By Lemma 4.3.2(1) `initTS` is only called on a pointer that has been in `VHead` previously, and `newN` has never been in `VHead` before this CAS, we know that `newN->ts` is `TBD`. So the invariant holds after the CAS, since `VHead->nextv->val = head->val`.

Now, consider a step s that is the linearization point of a modifying `vCAS(oldV, newV)`, which we denote V, possibly followed by the linearization points of some other `vCAS` operations that return `false` on Line 48. Since V is a modifying `vCAS`, it added a new `VNode` n_1 to the head of the version list in front of node n_2. This happens after V checks that n_2`.val = oldV` $\neq$ `newV` on Line 39–40 and sets n_1`.nextv` to point to n_2 and sets n_1`.val` to `newV` on Line 41. By Lemma 4.3.4, n_1 is still the head of the version list when step s occurs. So in the configuration C before s, the value in the versioned CAS object is n_2`.val = oldV`, by our assumption that the claim holds in C. Thus, when V occurs in the sequential execution, it returns `true` and changes the value of the versioned CAS object to `newV`. Note that `VHead->val = newV` in C'. It remains to check that all other `vCAS` operations that return `false` at line 48 and are linearized immediately after V should return `false` in the sequential execution and therefore do not change the value of the versioned CAS object. Consider any such `vCAS` V' of the form `vCAS(oldV',newV')`. By the definition of the linearization point of V', V makes the first change to `VHead` after V' reads it on

Line 37. So, V' must have read a pointer to n_2 on Line 37. Since V' returns `false` at Line 48, it must have seen n_2.`val = oldV'` at Line 39. Thus, `oldV' =` n_2.`val = oldV` $\neq$ `newV`, so when each of the `vCAS` operations V' is executed sequentially in linearization order, it should return `false` and leave the state of the versioned CAS object equal to `newV`. The claim for C' follows.

Finally, consider a step s that is the linearization point of one or more `vRead` operations or `vCAS` operations that return at Line 39 or 40. Consider any such operation op. Let h be the node at the head of the version list when op reads `VHead` at Line 33 or 37. Then s is either this **read** or a subsequent execution of Line 24 that makes h's timestamp valid. Either way, `VHead` points to h in C', by Lemma 4.3.4. By Lemma 4.3.5, either case (1) or (2) of the claim applies to configuration C. Either way, the value of the versioned CAS object in C is $h.val$. If op is a `vRead`, then it returns $h.val$ as it should. If op is a `vCAS` that returns `false` at Line 39, it would do the same in the sequential execution in linearization order because op reads the state of the versioned CAS object in C' from h.`val` on Line 39 and sees that it does not match its `oldV` argument. If op returns `true` at Line 40, it would also return `true` when performed in linearization order because the state of the versioned CAS object in C' matches both op's `oldV` and `newV` values. In all cases the value of the versioned CAS object does not change as a result of op, so it is still h.`val` in C', and the invariant is preserved. □

The following observation follows directly from the way modifying `vCAS` operations are linearized.

Observation 1. *Consider a* `VNode` *n that was added to the version list by a modifying* `vCAS` *V. If the timestamp of n is valid, then n.*`ts` *stores the value of* `S.timestamp` *at the linearization point of V.*

The following key lemma asserts that version lists are properly sorted.

Lemma 4.3.7. *The modifying* `vCAS` *operations are linearized in the order they insert* `VNode`*s into the version list.*

Proof. Consider any two consecutive `VNodes` n_1 and n_2 in the version list, where n_1 is inserted into the list before n_2, and let V_1 and V_2 be the `vCAS` operations that inserted n_1 and n_2 to the list, respectively. Recall that the linearization point of a modifying `vCAS` is at the read of the timestamp (Line 23) of the `initTS` call that validates the timestamp on the `VNode` that this `vCAS` appended to the version list. In particular, a modifying `vCAS` is linearized after it inserts its `VNode` into the list (since `initTS` cannot be called on a `VNode` before it is inserted, by Lemma 4.3.2(1)), but before its `VNode` is assigned a valid timestamp on Line 24 of `initTS`. By Lemma 4.3.3, a `VNode` is assigned a valid timestamp before it is replaced as the head of the version list. That is, V_1 must be linearized before n_1's timestamp was valid, and n_1's timestamp became valid before n_2 was added to the list. Furthermore, V_2 was linearized after n_2 was added to the list. Therefore, V_1 is linearized before V_2. □

Now, we prove our main theorem which says that our versioned CAS and camera algorithms are linearizable and have the desired time bounds.

Proof (Theorem 4.2.1). We show that the return values of each operation is correct with respect to their linearization points. For `vCAS` and `vRead` operations, this follows from Lemma 4.3.6.

We prove this for both `takeSnapshot` and `readVersion` simultaneously. Suppose that an S.`takeSnapshot` operation T returns a timestamp t, which is used as the input parameter of an O.`readVersion` operation R. We show that R returns the value of O at the linearization point of T. Let h be the value of `VHead` on line 27 of R. The timestamp of h is valid after line 28 of R, and by Lemma 4.3.3, the timestamps of all the nodes in the version list starting from h are valid. This means that on line 30, `node->ts` is never `TBD`. Let n be the value of `node` at the last line of R and let V be the modifying `vCAS` operation that appended n. We know that n is the first node in the version list starting from h with timestamp less than or equal to t. Since T is linearized when S.`timestamp` gets incremented from t to $t + 1$, by Observation 1, V is linearized before the linearization point of T. Since R returns the value written by V, it suffices to show that no modifying `vCAS` operation gets linearized between the linearization points of V and T. By Lemma 4.3.7, modifying `vCAS` operations are linearized in the order they appended `VNodes` to the version list. Therefore, for all nodes that are older than n in the version list, their modifying `vCAS` operations are linearized before the linearization point of V. Next, we show that all nodes in the version list that are newer than n are linearized after T. From the while loop on line 30, we can see that all nodes that lie between h and n (including h, excluding n) have timestamps are larger than t. All nodes in the version list that are newer than h also have timestamp larger than t because they are appended after line 27 of R and `S.timestamp` is already greater than t at this step. Therefore, by Observation 1, all nodes in the version list newer than n are linearized after the linearization point of T. This means V is the last modifying `vCAS` operation to be linearized before the linearization point of T, as required.

The bounds on the step complexity of the operations can be obtained by inspection of the pseudocode. □

4.4 Supporting Linearizable Wait-free Queries

We use versioned CAS objects to extend a large class of concurrent data structures that are implemented using reads and `CAS` primitives to support linearizable wait-free queries. Our approach is general enough to allow transforming many multi-point read-only operations on a sequential data structure into linearizable queries on the corresponding concurrent data structure. To achieve this, we define the concept of a *solo* query, i.e., a query that only reads the shared state, and once invoked, is correct if run to completion without any other process taking steps during its execution. Typically, solo queries can be implemented by adapting standard sequential queries.

The approach works as follows. Each `CAS` or `read` on a CAS object is replaced by a `vCAS` or `vRead` (respectively) on the corresponding versioned CAS object, all of which are associated with the same camera object. To perform a solo query operation q, a process p first executes `takeSnapshot` on the camera object, to obtain a handle ts. Then, for any CAS object that q would have accessed in the data structure, p performs `readVersion`(ts) on the corresponding versioned CAS object. Intuitively, `takeSnapshot` takes a snapshot of shared state, and solo queries then run on this snapshot while other threads may be updating concurrently.

Not all concurrent data structures can support solo queries. Herlihy and Wing [88] describe an array-based queue implementation in which the linearization order of the enqueue operations depends on future dequeue operations. For that algorithm, it is not possible to implement a solo query returning the state of the queue because the ordering between completed enqueues can be undetermined. However, for most data structures it is straightforward to implement solo queries. Here we give examples of several concurrent data structures that support solo queries. A thorough treatment of the conditions under which solo queries are possible and a comparison of these conditions with strong linearizability [76] are provided in [164].

FIFO Queue. We first consider Michael and Scott's concurrent queue (`MSQ`) [118], which supports atomic enqueues and dequeues, as well as finding the oldest and newest elements. Our scheme additionally provides an easy atomic implementation of more powerful operations such as returning the i-th element, or all elements, etc. The mutable locations in a `MSQ` consist of a `head` pointer, a `tail` pointer, and the `next` pointer of each node in a linked list of elements, pointing from oldest to newest. The `head` points indirectly to the oldest remaining element, and the `tail` points to the newest element, or temporarily to the element immediately before the newest. The newest element always has a null `next` pointer. After applying our approach, all these pointers become `vCAS` objects, and a `takeSnapshot` operation, T, will atomically capture the state of all of them. Any query can then easily reconstruct the part of the queue state it requires. For example, the `select(i)` query can start at the `head` and follow the list (calling `readVersion` on each node, using the handle returned by T) until it reaches the i-th element in the queue. Each `next` pointer in the linked list is only successfully updated once, so each `readVersion` of a `next` pointer takes constant time. Therefore, for example, finding the i-th element (from the head) in a queue takes time $O(i + c)$ where c denotes the number of successful dequeues between the read of the timestamp by T and the read of the `head`.

Sorted Linked List. Harris's data structure [80] maintains an ordered set as a sorted linked list (`HLL`), and supports insertions, deletions, and searches. Our approach adds atomic versions of multi-point query operations, such as range queries, finding the first element that satisfies a predicate, or multi-searches (i.e., finding if all or any of a set of keys is in the list). To implement concurrent insertions and deletions properly, `HLL` marks a node before splicing it out of the list. The mark is kept as one bit on the pointer to the next list node. Deletes are linearized when the mark is set. The mutable state comprises the `next` pointers of each link, which contains the mark bit. If these are versioned, a `takeSnapshot` captures the full state. A query can then follow the snapshotted linked list from the head, using `readVersion` on each node; all marked nodes should be skipped.

Time bounds for range query, multisearch and finding the i-th element were given in Table 1.1 in Section 1.2. Each insert or delete performs up to two successful `vCAS` operations and each successful `vCAS` may cause a query to traverse an extra version node. So, in the worst case, queries incur an additive cost of c (defined in Table 1.1). Each query also incurs an additive cost of P since it may encounter up to P marked nodes.

Binary Search Trees. We now consider concurrent binary search trees (BST). Many such data structures have been designed [7, 15, 29, 33, 36, 38, 62, 152, 170]. All the BST structures we looked into work with solo queries allowing for multi-point queries of the same type as in `HLL` (e.g., range queries and multisearches), but potentially much faster since they can often visit a small part of the tree. Queries on the structure of the tree (e.g., finding its height) can also be supported. Here we consider two such trees (which are also used in our experiments in Section 4.9): the non-blocking binary search trees (NBBST) of Ellen *et al.* [62], and the balanced non-blocking chromatic tree (CT) of Brown *et al.* [38].

The NBBST data structure is an unbalanced BST with the data stored at the leaves and the internal nodes storing keys for guiding searches. Every insertion involves inserting an internal node and a leaf, and similarly a delete removes an internal node and a leaf. The data structure uses a lock-free implementation of locks, "locking" one or two nodes for each insertion or deletion. The locks are implemented by pointing to a descriptor of the ongoing operation, so other threads can help complete the operation if they encounter a lock. This makes the data structure lock-free. The linearization point is at the pointer swing that splices an internal node (along with a child) in or out. Therefore at any time the child pointers of the internal nodes fully define the contents of the data structure. If these child pointers are kept as versioned CAS objects, then a snapshot will capture the required state. The queries can ignore the locks and, therefore, the descriptor pointers, although mutable, do not need to be versioned.

The chromatic tree (CT) is a balanced BST that also stores its data at the leaves. It is based on a relaxed version of red-black trees, with colors at each node facilitating rebalancing. Concurrent updates are managed similarly to the NBBST. In particular, updates are linearized at a single CAS that adds or removes a key. So, obtaining a snapshot of the tree's child pointers is sufficient to run multi-point queries.

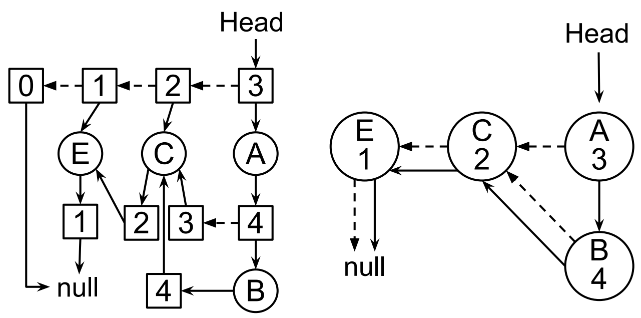
Any query q on NBBST or CT takes time proportional to the number of nodes it visits plus the write contention of q (i.e., the number of vCAS operations concurrent with q on memory locations accessed by q). This assumes q performs `readVersion` on each versioned CAS object at most once. This can be ensured by caching values read from the tree. For the bounds in Table 1.1, it suffices to show that the number of vCAS operations concurrent with q is at most the number of inserts, deletes and rotations concurrent with q. This is because each vCAS is either due to a rotation (only applies to CT) or is the linearization point of an insert or delete.

Importantly, our snapshot approach maintains the time bounds of all the operations supported by the original data structure. (For example, in the case of NBBST and CT, the original operations would be insert, delete, and lookup).

4.5 Avoiding Indirection

In this section and the next, we present ways to optimize our snapshotting approach (and therefore multi-point queries using such snapshots). We present these optimizations in terms of a concurrent data structure D to which we add snapshots and use them to run queries from a set Q. We denote by D' the version of D that also supports the queries in Q. Only the CAS objects in D that could potentially be read by queries need to be versioned in D'.

Algorithm 4.1 has a level of indirection even when accessing the most recent version of a vCAS object since it requires first accessing the head of the version list, and then the object it points to. Figure 4.2(a) illustrates an example of a linked list updated as described in Algorithm 4.1 along with its version lists. Here we discuss how this indirection can be avoided. Figure 4.2(b) illustrates the linked list after applying the optimization (more details later). This optimization has some restrictions, which we define first.



(a) Indirect vCAS (b) Direct vCAS

Figure 4.2: A simple concurrent linked list using both direct and indirect versioned CAS objects. The state of each results from inserting keys *E*, *C*, *A*, and *B* (in that order) into an empty list. Circles represent linked list nodes and squares represent `VNodes` from Algorithm 4.1. Numbers represent timestamps, dotted arrows represent version pointers (`nextv` pointers), and solid arrows represent the value field `val`.

```
class Node {
  /* other fields of the Node class */
  int ts;   // initially TBD
  Node* nextv; };
class DirectVersionedCAS {
  Node* Head; Camera* S;
  DirectVersionedCAS(Node* n,
                     Camera* s) {
    Head = n; S = s; initTS(n); }
  void initTS(Node* n) {
    if(n != NULL && n->ts == TBD) {
      int curTS = S->timestamp;
      CAS(&(n->ts), TBD, curTS); } }
  Node* vRead() {
    Node* head = Head;
    initTS(head);
    return head; }
```

```
Node* readVersion(int ts) {
  Node* node = Head;
  initTS(node);
  while(node != NULL &&
             node->ts > ts)
    node = node->nextv;
  return node; }
bool vCAS(Node* oldV, Node* newV){
  Node* head = Head;
  initTS(head);
  if (head != oldV) return false;
  if (newV == oldV) return true;
  newV->nextv = oldV;
  if(CAS(&Head, head, newV)) {
    initTS(newV);
    return true; }
  else {
    initTS(Head);
    return false; } } };
```

Algorithm 4.3: Linearizable implementation of a versioned CAS object without indirection.

A *versioned node* is a node that versioned CAS objects can point to directly. We say that a versioned node is *recorded* in a history H if a pointer to it is the newV parameter of a *successful* vCAS (on any versioned CAS object) in H. When a node is recycled by the memory reclamation scheme, it counts as a new object.

Definition 3 (Recorded-once)**.** *D' is* recorded-once *if, for every history of D', the following hold: (1) every versioned node is recorded at most once, (2) the* newV *parameter of each* vCAS *is a pointer to a versioned node, (3)* vCAS *operations (both successful and unsuccessful) with the same* newV *parameter must have the same* oldV *parameter, and (4) versioned CAS objects are initialized with null or a pointer to a previously recorded node.*

This property allows us to overload a versioned node as both a node and a link in a version list, thus avoiding the indirection. Conditions 2 to 4 are relatively natural to satisfy, so it is Condition 1 that is most important.

Our approach works as follows. For each versioned CAS object O that stores a pointer to a node in D', instead of creating a new VNode to store the version pointer and the timestamp, we store this information directly in the node pointed to by O. To do this, we extend each node with two extra fields, ts and nextv, and modify Algorithm 4.1 accordingly. The resulting algorithm is described in Algorithm 4.3 and we call it the *direct* implementation of versioned CAS objects. Naturally, we call Algorithm 4.1 the *indirect* implementation. Figure 4.2 gives an example using both versions.

The correctness of Algorithm 4.3 depends heavily on the recorded-once property (Definition 3). Condition 1 ensures that every versioned node appears as a non-tail element of a version list at most once. Note, however, that a versioned node x can appear as the tail of the version lists of an arbitrary number of vCAS objects since each can set their initial value to x. In the example in Figure 4.2, the node C is both the tail of the version list from B and an internal node for the version list from the head. A timestamp is set on a versioned node when it is recorded (Line 32, or by someone helping), and by Condition 1 this means it is set at most once. Furthermore, by Condition 4, a node is not used as an initial value until its timestamp is set, meaning that all timestamps stored in D' are already set or in the process of being set (possibly by helping). Condition 2 is required to ensure we have somewhere to record the timestamp and next node in the version list (i.e., newV needs nextv and ts fields). Condition 3 ensures that all vCAS operations with the same newV attempt to write the same pointer into newV->nextv on Line 30.

The direct implementation can be applied to concurrent data structures for which, at any point in time, every object has at most one pointer to it. Examples include tree data structures where pointers go from parents to children, or singly-linked lists. However, this can involve slight modifications to the original concurrent algorithm. For example, if a node being pointed to by one object is moved to be pointed to by another object then it would be recorded more than once. This can happen during a **delete** operation in HLL [80] and NBBST [62]. To avoid this, the node can be copied and a pointer to the new node can be written into the new location. This modification should be done with care to preserve correctness. We apply this transformation in our NBBST implementation (Section 4.8).

4.6 Other Optimizations

In this section, we present additional optimizations that work for both the direct versioned CAS algorithm from Section 4.5 and the indirect algorithm from Section 4.2.

Removing redundant versions. If snapshot operations are infrequent, many consecutive nodes in a version list may have the same timestamp. Since only the most recent such node is needed by `readVersion`, we can save space by storing only nodes that have distinct timestamps. This can be done by modifying the `vCAS` operation slightly. After setting the timestamp of a newly added version list node (i.e., after Line 43 of Algorithm 4.1), we modify `vCAS` to splice out the next node in the version list if it has the same timestamp as the head of the version list. We also modify `vCAS` to perform this splicing step before attempting to append a new node (i.e., before Line 42 of Algorithm 4.1), and this ensures all nodes in a version list, except possibly the two most recent ones, have distinct timestamps. When using this optimization in Algorithm 4.3, Line 30 should be modified to update `nextv` with a `CAS` so that this optimization does not get undone.

Avoiding contention. Although `takeSnapshot` only uses a single CAS, this CAS can be highly contended if snapshots are frequently taken. To reduce contention, we observe that C.`takeSnapshot` must only ensure that C's timestamp is incremented at some point during its execution interval. Thus, C.`takeSnapshot` can use exponential backoff to wait for another process to increment C's timestamp. After waiting, if no process has done so, then C.`takeSnapshot` tries to do the increment itself.

4.7 Memory Reclamation

To add memory reclamation to our snapshot approach, we use Epoch Based Memory Reclamation (EBR) [71]. EBR splits an execution into epochs by utilizing a global epoch counter `EC` (with initial value 1). Interestingly, with our direct implementation of versioning we are able to collect exactly the same nodes as can be collected in non-versioned EBR—i.e., all nodes that were freed prior to the last two epochs. There can still be some additional memory overhead for versioning, however, due to the extra `nextv` and `ts` field in each node, and, as mentioned in Section 4.5, the need in some algorithms to allocate extra nodes when deleting.

EBR supports three operations, `BeginOp`, `EndOp` and **retire**. A process p executing a `BeginOp` operation simply reads `EC` and announces the value read as its current epoch. An `EndOp` by p clears any previously announced epoch by p. EBR maintains a per-process *limbo list* of objects for each epoch. An object is added to the limbo list of the most recent epoch whenever it is passed to **retire**. When all processes have announced an epoch number that is at least b (where b is any integer greater than 2), the limbo list associated with epoch $b-2$ is collected, and the global epoch counter, `EC`, is incremented. In this way, EBR maintains only the limbo lists of the last three epochs. For our experiments, we use an efficient variation of EBR called DEBRA [39].

Using the notation of Section 4.5, let D be a concurrent data structure and let D' be the snapshottable version of D that supports a set of query operations Q in addition to the operations supported by D. In languages *without* automatic garbage collection, we can support memory reclamation for D' as follows. `BeginOp` is invoked at the beginning of each operation, and `EndOp` is invoked at the end. **retire** is called on a node whenever it is removed from the current version of the data structure. If D' uses the indirect implementation from Algorithm 4.1, before returning from a successful `vCAS` on Line 44, we also have to **retire** the `VNode` pointed to by local variable `head`. Furthermore, when a data structure node y is retired, we have to **retire** all the `VNodes` at the head of y's versioned CAS objects. The key observation is that all operations from D' (including query operations) only access nodes that were in the current version of the data structure at some point during the operation's execution interval. This means that whenever EBR determines that a node is retired before the start of the earliest live operation, we can free the node without first unlinking it from any version list because it can no longer be accessed by any live operation.

In languages *with* automatic garbage collection, we first use EBR to unlink nodes from version lists and then rely on the garbage collector to clean up any unreachable nodes. We modify the EBR algorithm so that when a limbo list is cleared and its nodes reclaimed, for each node x in the limbo list, instead of freeing x, we set its version list pointer, x`->nextv`, to null. We can think of this as retiring a version list pointer rather than a node. A query operation q working on a snapshot of the data structure protects any version list pointers it may access by calling `BeginOp` before taking a snapshot and calling `EndOp` when it is done using the snapshot. Since q can access version list pointers only of those nodes that were added during q's execution interval, it is safe to retire a version list pointer as soon as the pointer is added to the data structure. EBR ensures that this pointer is not set to null until all operations that were live when it was retired terminate. This means that in Algorithm 4.1 (Algorithm 4.3), we retire the pointer `newN->nextv` (`newV->nextv`, respectively) before returning from a successful `vCAS` on Line 44 (Line 33, respectively).

4.8 Implementation

We implemented our snapshotting approach in both Java and C++. Using the implementations, we then implemented snapshottable versions of three existing lock-free external BST data structures (see details below). We use all the optimizations discussed in Sections 4.5 and 4.6. To apply our approach on top of these tree data structures, we make each node in the data structure *versionable* by adding a timestamp and a version pointer field to it, use direct versioned CAS objects for child pointers, and modify the data structure to be recorded-once if necessary. All versioned CAS objects are associated with the same camera object so we avoid storing a pointer to a camera object in each versioned CAS object.

The key and value fields of each node are immutable in all three tree data structures. So, a snapshot of the child pointers completely defines the contents of the tree and can be used to answer arbitrary multi-point queries. For our implementation in Java, we implemented the four queries in Table 4.1, and for our implementation in C++, we implemented range queries. All are linearizable. We reclaim memory using the EBR based technique described in Section 4.7. Our code is publicly available on GitHub[2].

Base Data Structures. We applied our snapshotting approach to the two BSTs described in Section 4.4, `NBBST` and `CT`, and to a lock-free unbalanced BST designed using Brown, Ellen and

[2]`https://github.com/yuanhaow/vcaslib`

Query	Definition	Parameters in Figure 4.4m
`range`(s, e):	All keys in range $[s, e]$	`range256`: $e = s + 256$
`succ`(k, a)	The first a key-values with key greater than k	`succ1`: $a = 1$, or `succ128`: $a = 128$
`findif`(s, e, f) [47]:	The first key-value pair in range $[s, e)$	`findif128`: $f(k) = (k \ mod \ 128 \ is \ 0)$
`multisearch`(L):	For a list of keys in L, return their values (null if not found)	`multisearch4`: $\|L\| = 4$

Table 4.1: The multi-point queries and their parameters we use in experiments.

Ruppert's Tree Update Template [38]. For the first two, we used Brown's Java implementations [35]. For the third, we used the C++ implementation by Arbel-Raviv and Brown [7].

Batching. Previous work has shown that the performance of concurrent BSTs is improved by batching keys. We therefore applied the same batching technique from `PNB-BST` [67] and `LFCA` [170] to our Java implementations, storing up to 64 key-value pairs in each leaf (see [67] for more details). We did not apply batching in our C++ code since it was also not used by the C++ implementation [7] we compared with.

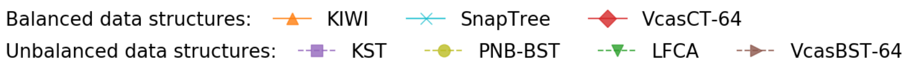
Recorded-Once. The recorded-once requirement is naturally satisfied by `CT` and the BST from [7], but not by `NBBST` because the delete operation uses `CAS` to swing a pointer to a node that is already in the data structure. To avoid this, our implementation copies the node and swings the pointer to this new copy instead. This requires some extra marking and helping steps to preserve correctness and lock-freedom.

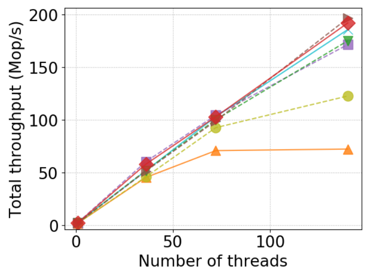
Names. `BST-64` and `CT-64` are the non-snapshottable Java BSTs (with batching). `VcasBST-64` and `VcasCT-64` are our snapshottable versions. `BST` is the non-snapshottable C++ BST from [7], while `VcasBST` is our snapshottable version.

4.9 Experimental Evaluation

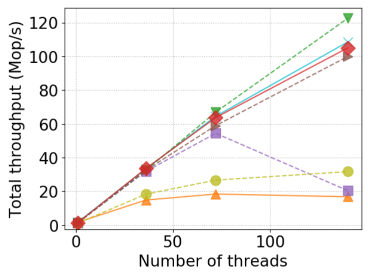
In this section, we provide our experimental analysis, which has two main goals: first, to understand the overhead that our approach introduces to concurrent data structures which originally did not support multi-point queries, and second, to compare the performance of our approach to that of state-of-the-art alternatives which support atomic range queries.

Other Structures that Support Range Queries. We compare with several state-of-the-art dictionary data structures: `SnapTree` [33], `KiWi` [15], `LFCA` [170], `PNB-BST` [67], `KST` [36], and `EpochBST` [7] using code provided by their respective authors. Arbel-Raviv and Brown [7] presented several ways to add range queries to concurrent data structures. We use `EpochBST` to refer to their most efficient range queryable lock-free BST, which is implemented in C++. This serves as a good comparison for `VcasBST` because they both augment the same initial BST with linearizable range queries. All the other data structures are written in Java. They are all lock-free except `SnapTree`, which uses fine-grained locking. We classify `KiWi`, `SnapTree`, and `VcasCT-64` as *balanced* data structures because they have logarithmic search time in the absence of contention, and the others as *unbalanced*. For the k-ary tree (`KST`), we use $k = 64$ which was shown to perform well across a variety of workloads [36]. We used batch size 64 for

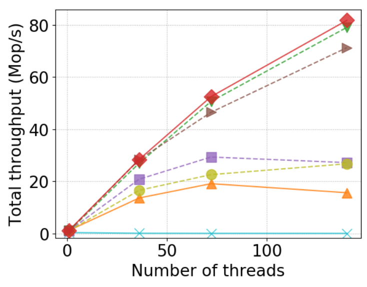




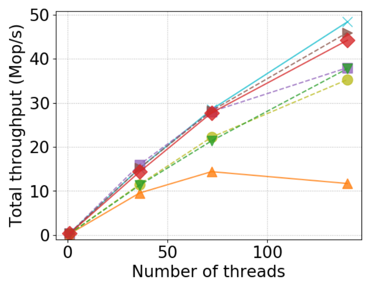
(a) Lookup heavy - 100K Keys: 3%ins-2%del-95%find-0%rq



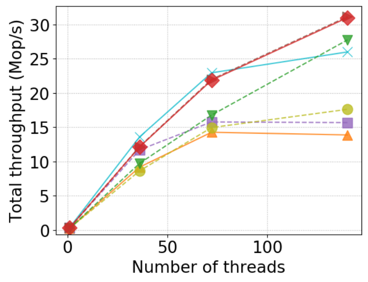
(b) Update heavy - 100K Keys: 30%ins-20%del-50%find-0%rq



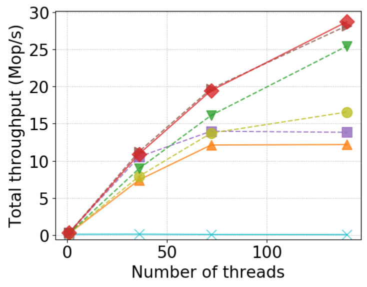
(c) Update heavy with RQ - 100K Keys: 30%ins-20%del-49%find-1%rq-1024size



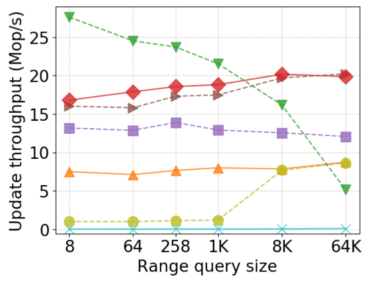
(d) Lookup heavy - 100M Keys: 3%ins-2%del-95%find-0%rq



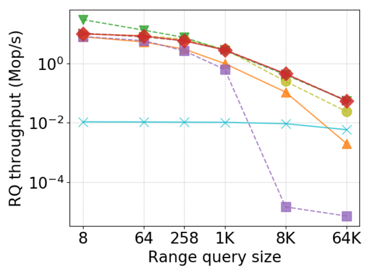
(e) Update heavy - 100M Keys: 30%ins-20%del-50%find-0%rq



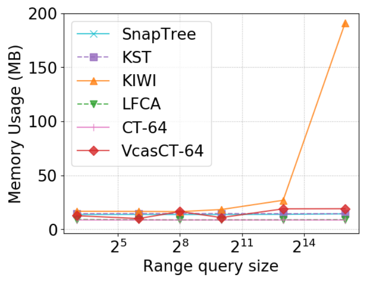
(f) Update heavy with RQ - 100M Keys: 30%ins-20%del-49%find-1%rq-1024size



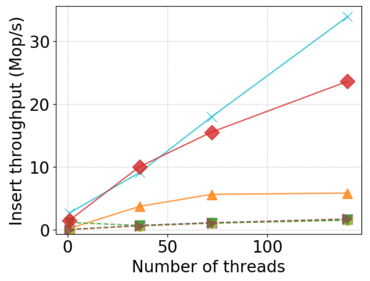
(g) Update Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads



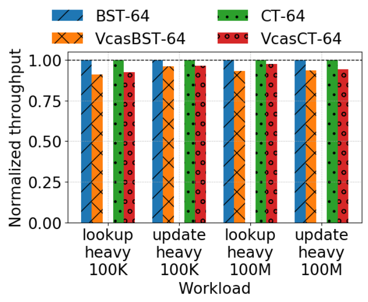
(h) RQ Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads



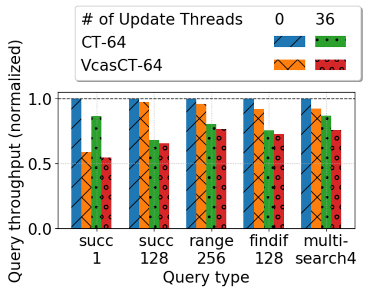
(i) Memory Usage - 100K keys: 36 Update Threads, 36 RQ Threads



(j) Insert Only, Sorted Sequence

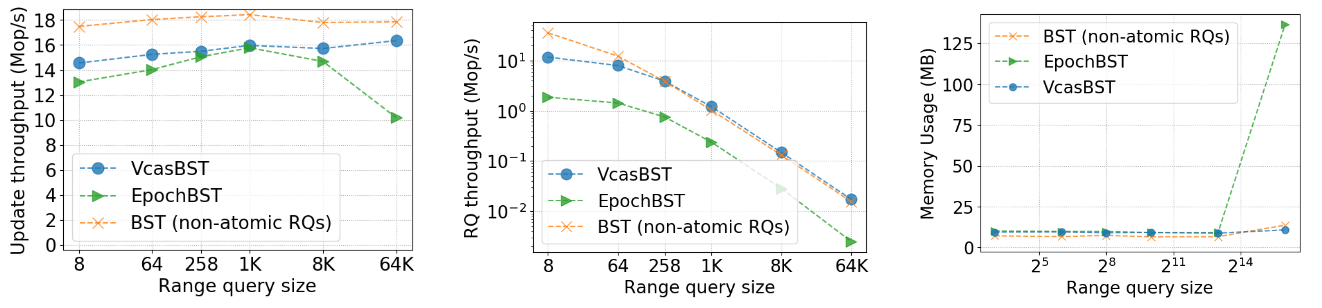


(k) Overhead of Vcas, 140 threads measured across various workloads



(m) Throughput of atomic queries on VcasCT-64

Figure 4.4: Java experiments.



(a) Update Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads (b) RQ Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads (c) Memory Usage - 100K keys: 36 Update Threads, 36 RQ Threads

Figure 4.5: C++ experiments.

VcasBST-64 and VcasCT-64, as well as for LFCA and PNB-BST. This batch size has been shown to yield good range query performance for LFCA and PNB-BST in [67, 170].

We also applied the contention avoiding technique from Section 4.6 to KiWi, PNB-BST, and EpochBST because we found that it also improved their performance by reducing contention on the global timestamp.

Setup. Our experiments ran on a 72-core Dell R930 with 4x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. We used `numactl -i all`, evenly spreading the memory pages across the sockets in a round-robin fashion. The machine runs Ubuntu 16.04.6 LTS. The C++ code was compiled with g++ 9.2.1 with `-O3`. Jemalloc was used for scalable memory allocation. For Java, we used OpenJDK 11.0.5 with flags `-server`, `-Xms300G` and `-Xmx300G`. The latter two flags reduce interference from Java's GC. We report the average of 5 runs, each of 5 seconds. For Java we also pre-ran 5 runs to warm up the JVM. The variance is small in almost all tests.

Workload. We vary four parameters: data structure size n, operation mix, range query size *rqsize*, and number of threads. In most experiments, we prefill a data structure with either $n = 100K$ or $n = 100M$ keys. These sizes show the performance when fitting and not fitting into the L3 cache. Keys for operations, and in the initial tree, are drawn uniformly at random from a range $[1, r]$, where r is chosen to maintain the initial size of the data structure. For example, for $n = 100K$ and a workload with 30% inserts and 20% deletes, we use $r = n \times (30 + 20)/30 \approx 166K$. We perform a mix of operations, represented by four values, *ins*, *del*, *find*, and *rq*, which are the probabilities for each thread to execute an `insert`, `delete`, `find`, and `range`, respectively. Unbalanced trees can be balanced in expectation using uniformly random keys, so we also run a workload with keys inserted in sorted order.

Scalability. Figures 4.4a-4.4f show scalability (in Java) under a variety of workloads. Note that in Figures 4.4c and 4.4f, although range queries are only performed with 1% probability, they occupy a significant fraction of execution time.

Generally, VcasCT-64 and VcasBST-64 (our two implementations), and LFCA have the best (almost-linear) scalability across all workloads. LFCA outperforms our implementation in Figure 4.4b, but it is consistently slower in the 100M-key experiments (Figures 4.4d-f). SnapTree is competitive with our trees in the absence of range queries, but it has no scalability with range queries due to its lazy copy-on-write mechanism. Overall, VcasCT-64 is always among the top three algorithms and in most cases has the best performance.

Varying Range Query Size. We show the effect of varying range query size in Figures 4.4g and 4.4h (Java), and Figures 4.5a and 4.5b (C++), in which 36 dedicated threads ran range queries and 36 ran updates. Each update thread performs 50% inserts and 50% deletes on a data structure initialized to 100K keys. To better understand the cost of updates and range queries, we plot the throughput of each operation separately.

In Figure 4.4g, PNB-BST has low update throughput when $rqsize \leq 1024$. This is because its update operations are forced to abort and restart whenever a new range query begins, and thus more frequent range queries lowers update throughput. KST performs decently in most workloads except when each range query covers a significant fraction of the key range. This is because their range query performs a double collect of the desired range and is forced to restart if it sees an update between the two collects.

Data structures that increment a global timestamp with every range query become bottlenecked by this increment when range queries are frequent. This applies to our trees as well as PNB-BST, KiWi, and EpochBST. Consequently, with $rqsize = 8$, LFCA has 3x faster range queries when compared to our trees (Figure 4.4h). However, LFCA avoids using a global timestamp by having update operations help ongoing range query operations. This helping becomes more frequent and more costly when *rqsize* is large, as shown in Figure 4.4g. For $rqsize = 64K$ (about a third of the key range), the update throughput of our trees is 4x faster than LFCA. Other than LFCA, all the other implementations have mostly stable update throughput with varied range size, among which VcasCT-64 has the best overall performance.

Figure 4.5 compares the performance of the C++ version of VcasBST with that of EpochBST. Range queries on VcasBST are 5–7x faster than EpochBST. This is because EpochBST effectively maintains a single global version list rather than a separate version list for each object. So range queries on EpochBST are slowed down by all concurrent updates, even ones that are outside of the range being queried for, because they increase the cost of searching the global version list. In contrast, a range query in VcasBST is only slowed down by updates that are within the query range. In terms of update performance, VcasBST is at least as fast as EpochBST, and up to 60% faster on the largest range query size.

Sorted Workload. In Figure 4.4j, we test the Java implementations under a sorted workload. We insert an array of sorted keys into an initially empty tree by splitting the array into chunks of size 1024 and placing the chunks on a shared work queue; when a thread runs out of work, it grabs a new chunk from the head of the shared work queue. As expected, the balanced trees, VcasCT-64, KiWi and SnapTree, outperform the unbalanced ones. On 140 threads, SnapTree is 1.4x faster than VcasCT-64, which is in turn 4.1x faster than KiWi.

Overhead of Our Approach. In Figure 4.4k, we compare the throughput of our Java implementations VcasBST-64 and VcasCT-64 with the original data structures, BST-64 and CT-64, using 140 threads. The numbers in Figure 4.4k are normalized to the throughput of BST-64 and CT-64 to make the overheads easier to read. The overall overhead of our approach is low, ranging between 2.7% and 9.1% depending on the workload. This overhead includes the time for epoch-based memory management and the cost of using `vCAS` and `vRead`. For `VcasBST-64`, it also includes the actions we take to ensure that deletes satisfy the recorded-once property. The overhead is low because a `vRead` rarely sees a node with timestamp `TBD`, and therefore rarely performs a CAS to help set the timestamp. While a `vRead` sometimes incurs a cache miss when checking if this helping step is required (Line 11 of Algorithm 4.3), in the data structures we test, this cache miss would have been incured anyways when the operation follows the pointer returned by the `vRead`.

We also measure the overhead of our multi-point queries, `range`, `multisearch`, `succ`, and `findif`, with parameters shown in Figure 4.4m. We compare throughputs for `VcasCT-64` with *non-atomic* multi-point queries on the original `CT-64`, which simply run their sequential algorithms (and are not linearizable). Non-atomic `multisearch`, for example, simply calls `find` for each key. Figure 4.4m shows the cost that our approach has to pay to provide query atomicity.

All queries other than `succ1` exhibit low overhead: they are between 2.9% and 12.8% slower than their non-atomic counterparts. For `succ1`, our scheme exhibits larger overheads (36.8-41.4%) due to the bottleneck of incrementing the global timestamp when there are lots of small fast queries.

Memory Usage. Figures 4.4i and 4.5c show memory usage graphs for the Java and C++ data structures, respectively. In our Java experiments, we measured the amount of heap memory in use after Java's garbage collector cleans up all unreachable objects. We found that `VcasCT-64`'s memory usage is within a factor of 2.2 of both `CT-64` and `LFCA`, which tie for having the smallest memory footprints. We omitted `PNB-BST` from Figure 4.4i because it does not allow for garbage collection and uses significantly more memory than the rest.

For the C++ experiments, we measured memory usage by multiplying the number of allocated nodes by the size of each node. As discussed in Section 4.7, `VcasBST` has little memory overhead with respect to the non-versioned `BST` because they both use EBR and keep around approximately the same number of nodes. Most of the overhead comes from storing an extra timestamp and version pointer in each node.

Summary. Overall, our snapshot approach has low overhead and, despite its generality, performs well compared to existing special-purpose data structures. In particular, `VcasCT-64` had the best overall throughput among all the range queryable Java data structures we tested. `VcasBST-64` is also competitive on uniform workloads.

4.10 Conclusion and Discussion

In this chapter, we show a simple and efficient approach for snapshotting and supporting multi-point queries on a large class of concurrent data structures. Our approach maintains the time bounds and progress properties (e.g. lock-freedom/wait-freedom) of the original data structure, and it also ensures that the newly added multi-point queries are wait-free. Moreover, the technique provides good time bounds, with multi-point queries taking time proportional to their sequential complexity plus a contention term representing the number of update operations concurrent with the query. Despite its generality, it is also extremely efficient, often outperforming data structures specifically designed to support fast range queries.

This chapter focuses mostly on `CAS` based data structures, but these ideas can be extended to work for LL/SC based data structures as well.

Chapter 5

Versioned Pointers

5.1 Introduction

This chapter implements a new abstraction called *versioned pointers* which improves upon versioned CAS objects introduced in the previous chapter in three important ways. First, it contains a more automated mechanism for avoiding indirection that does not require converting the data structure to satisfy the recorded-once condition defined in Chapter 4.5. For some data structures, converting them to be recorded-once can be tricky and add additional cost. This chapter introduces a hybrid of the indirect and direct versioned CAS algorithms where indirection is used only when needed (i.e., when an object is recorded more than once) and uses shortcutting to remove the indirection when it is no longer needed. The user does not have to know anything about being recorded-once. We refer to this as *indirection on need.*

Second, although versioned CAS objects can be used with locks, it is not optimized for locks, and hence has some inefficiencies. When using locks, shared variables are typically updated with stores instead of CASes. Using the previous chapter's approach, a store can be implemented by loading the value and then a CAS with the loaded value as the expected value, but this requires several unnecessary steps. This chapter streamlines this by directly supporting a store operation on versioned pointers, and more importantly, converts the code to be idempotent so that versioned pointers can be used alongside lock-free locks from Chapter 3. Achieving idempotence required several new ideas. For example, the idempotence approach from Chapter 3 does not support CAS, which is required by both versioned pointers and versioned CAS.

Third, this chapter presents a mechanism for incrementing the global timestamp. As pointed out in Section 5.7, when snapshots are used for smallish queries this increment can be a significant bottleneck due to contention on the stamp. One solution to this is to use synchronous (across cores) hardware clocks for time stamps [96, 146]. However such clocks are not available on all machines and even on machines that do appear to support them, their properties regarding synchronization among nodes are not fully documented by the vendors [44]. This chapter presents a timestamping mechanism that optimistically runs queries without incrementing the time stamp. If the query runs into a timestamp that equals its own it aborts, increments the stamp and reruns (at most once). We refer to this as *optimistic timestamping*.

Based on these ideas we have developed and implemented an easy-to-use C++ library called VERLIB. The library revolves around a *versioned pointer* type, which can be used in either lock-based or CAS-based concurrent data structures to store a shared pointer. As with atomic locations in many programming languages, the versioned pointer supports atomic loads, stores, and CASes on the locations. The user can convert their existing concurrent data structure to use VERLIB with only a couple changes: (1) replacing atomic locations holding pointers that need to be part of the snapshotted state with versioned pointers, and (2) inheriting a "versioned" class into any objects pointed to by such pointers. Then the user can just wrap a collection of loads in the `with_snapshot` function provided by VERLIB and all loads will see an atomic view—i.e., the state of the versioned pointers at some fixed point in time within the scope of the `with_snapshot`.

Once a concurrent data structure is modified to use VERLIB, compiler flags can be set to run it in several different modes. It can be used as a standard concurrent data structure with no support for snapshots and no lock-free locks. In this case, the loads within a `with_snapshot` will still work, but not be atomic with each other. Also any locks will be blocking locks instead of lock-free locks. If versioning is turned on, the loads within `with_snapshot` will present a consistent snapshot of the data structure. If lock-free locks are turned on, the locks will be lock-free. The user can specify the type of timestamp they want, switching among QUERYTS, OPTTS, and HWTS. QUERYTS is the approach used in Chapter 4 while OPTTS uses optimistic timestamping and is typically significantly faster, but requires that the query can be rerun a second time. HWTS uses synchronous hardware timestamps and can only be run on machines that properly support them. Any combination of the above settings can be used.

We have converted several state-of-the-art concurrent data structures to use the approach, including a singly and doubly linked list, a hash table, an adaptive radix tree (ART) [104], and a b-tree. All but the hash table are taken from the FLOCK library from Chapter 3, and the hash table uses array bucket copying [48].[1] In the chapter, we present several experimental results comparing the different data structures, with the different settings of the flags mentioned above, and under a variety of workloads. The workloads include various mixes of updates (inserts and deletes), finds, range queries, and multi-finds. We also vary the data structure sizes and the skewness of the distribution using a zipfian distribution. We then compare performance to some existing data structures that directly support range queries.

The experiments show that the cost of versioning is typically small. They also show that indirection-on-need is much more efficient than using indirection, while not requiring any conversion to the code. The experiments show that optimistic timestamping significantly outperforms greedy increments on smallish queries, and is almost always nearly as fast as hardware timestamps (sometimes slightly faster). Finally they show that versioning integrates well with lock-free locks.

The contributions of the chapter include:

- A new *indirection-on-need* approach for version lists that mostly avoids indirection, while not requiring that objects are only recorded once.

[1]We believe our starting implementations are the fastest or competitive with the fastest current implementations for sorted lists, unsorted sets, radix-sorted sets and unsorted-sets.

- Efficient and full support of versioned pointers inside of both blocking and lock-free locks. This includes a new mechanism to support an idempotent CAS.
- A easy-to-use portable library, VERLIB, for adding versioning to existing or new concurrent data structures.
- The first b-tree we know of that is lock-free and versioned. It is also significantly faster than previous data structures designed to support linearizable range queries.
- The first versioned radix tree, whether lock-free or not.
- A collection of experiments demonstrating the various tradeoffs of our approaches.

5.1.1 Example: Doubly Linked List

Here we present an example of how the VERLIB interface can be used to extend the doubly-linked sorted list from Section 3.1.1 to support snapshots. The data structure uses lock-free locks. In addition to insertions, deletions and finds, supported by the prior data structure, the snapshots allow for atomic range queries and any other query involving a snapshot of the state of the list. We give the code for deletions and range queries in Algorithm 5.1 and code for find and remove in Algorithm 5.2 where changes from the original are marked in red.

Each `node` of the list holds a key and value, previous and next pointers, a lock, and a flag indicating whether the node has been removed. The `versioned_ptr` on (line 4) indicates that the `next` pointer should be versioned (i.e., used in an atomic snapshot). In VERLIB, the `versioned` class needs to be inhereted for any class `X` that is used as `versioned_ptr<X>` (line 1). Any classes that inherit `versioned` must also use the VERLIB epoch-based memory allocator (lines 7 and 21).

The `range` operation implements an atomic range query from key `k1` (inclusive) to key `k2` (exclusive). It finds the first key greater or equal to `k1` using `find_node`, and then continues traversing the list while pushing keys onto `result` until finding a key greater or equal to `k2`. We assume the list has a sentinel infinite key at the end. The `with_snapshot` takes as its only argument a thunk `f` (a lambda expression with no argument)[2] and runs it such that all its loads see an atomic view of the memory state (i.e. of all versioned pointers). The range query will therefore be atomic (i.e., linearizable with updates). Note that only the `next` pointer needs to be versioned since it is the only mutable shared variable read by the range query operation. Without the `with_snapshot` the code is not atomic—effects of updates that are concurrent with the query might or might not appear in the result.

The `insert` searches for the first node `next` with a key greater or equal to `k` and tries to acquire a lock on its previous node (`prev`). If the lock is successfully acquired, `prev` has not been removed, and `prev->next` still points to `next`, the algorithm allocates a new node and splices it in. Otherwise it makes another attempt by repeating the `while` loop. The `lck->try_lock(f)` is from the FLOCK library. It attempts to take the lock on `lck` and, if successful, runs the thunk `f`. It returns true if and only if the lock was successfully acquired and the thunk returned true. It is lock-free in the formal sense as defined in Chapter 2.

[2]In C++ "`[=] { body }`" creates a lambda with no arguments where the free variables of the body are captured by value.

```
struct node : verlib::versioned {
  Key k; Value v;
  flck::atomic<node*> prev;   // not versioned
  verlib::versioned_ptr<node> next;
  flck::lock lck;
  flck::atomic<bool> removed; }; // not versioned
verlib::memory_pool<node> nodes;
node* find_node(node* head, Key k) {
  node* cur = (head->next).load();
  while (k > cur->k) cur = (cur->next).load();
  return cur; }
std::vector<Key> range(node* head, Key k1, Key k2) {
  return verlib::with_snapshot([=] {
    std::vector<Key> result;
    node* cur = find_node(head, k1);
    while (cur->k < k2) {
      result.push_back(cur->k);
      cur = (cur->next).load(); }
    return result; }); }
bool insert(node* head, Key k, Value v) {
  return verlib::with_epoch([=] {
    while (true) {
      node* next = find_node(head, k);
      if (next->k == k) return false; // already there
      node* prev = (next->prev).load();
      if (prev->k < k &&
          prev->lck.try_lock([=] {
            if (prev->removed.load() || // validate
                (prev->next).load() != next)
              return false;
            node* cur = nodes.alloc(k, v, next, prev);
            prev->next = cur; // splice in
            next->prev = cur;
            return true;}))
        return true;}});} // success
```

Algorithm 5.1: Using verlib to extend Flock's sorted doubly-linked list to support atomic range queries. Changes are marked in red.

```
std::optional<Value> find(node* head, Key k) {
  return verlib::with_epoch([=] {
    node* cur = find_node(head, k);
    if (cur->k == k) return std::optional<Value>(cur->v);
    else return {}; });}

bool remove(node* head, Key k) {
  return verlib::with_epoch([=] {
    while (true) {
      node* cur = find_node(head, k);
      if (cur->k != k) return false; // not found
      node* prev = (cur->prev).load();
      if (prev->lck.try_lock([=] {
            return cur->lck.try_lock([=] {
              if (prev->removed.load() || // validate
                  (prev->next).load() != cur)
                return false;
              node* next = (cur->next).load();
              cur->removed = true;
              prev->next = next; // splice out
              next->prev = prev;
              nodes.retire(cur);
              return true;});}))
        return true; } }); } // success
```

Algorithm 5.2: Find and Remove for doubly-linked lists. Extends Algorithm 3.1.

```
template <typename T>
struct versioned_ptr {
  versioned_ptr(T v);  // constructor with value v
  T load();            // read the value
  void store(T v);     // store a new value
  bool cas(T old_v, T new_v); // compare and swap
  T operator=(T v) {   // overload assignment with store
    store(v); return v; } };
```

Figure 5.3: Interface for a `versioned_ptr` in VERLIB.

5.2 Background

Chapter 3 describes a library, Flock for supporting lock-free locks. Converting lock-based codes into Flock requires replacing atomic shared variables with the `flck::atomic` version, which then implements idempotence. A `flck::atomic` supports a `load`, `store` and `cam` operations. In some cases in this chapter, for efficiency, we use non-idempotent versions of `load` and `cas`, which will be denoted as `load_non_idempotent` and `cas_non_idempotent`. These are implemented by primitive load and CAS respectively. Also Flock supplies an `atomic_write_once` which can be used if the location is only written to once after it is initialized and is slightly more efficient than `flck:atomic`.

5.3 VERLIB

Here we present the rather minimal VERLIB interface. Although presented and implemented in C++, it should not be hard to embed the ideas in libraries for other programming languages. The library consists of the following two classes.

- A `versioned_ptr<T>` class which is used to store versioned pointers to objects of type `T`. It supports the operations described in Figure 5.3.
- A `versioned` class that must be inherited by every type `T` that is used in a `versioned_ptr<T>`. It has no user accessible fields.

The library also supports the function:

- `with_snapshot`(f), which takes a thunk f and runs it such that all calls to `load()` on versioned pointers return values at a fixed point in the linearized order of updates which falls between the invocation and response of the `with_snapshot`. It returns the value returned by f.

Finally the user needs to use the VERLIB memory manager. Versioned objects (that inherit `versioned`) must be managed through an object `flock::memory_pool<T>` which supports the two operations: `alloc(...args)`, which given constructor arguments `args` allocates a new object of type T and returns a pointer to it, and `retire(T* ptr)`, which retires the object pointed to by `ptr`. Furthermore all concurrent operations must be wrapped in `flock::with_epoch`(f).

Examples of how to use all these were given in Section 5.1.1. This interface is common across all our implementations. If the structure is to be used with lock-free locks (not required) then

```
struct versioned {}; // empty for the indirect variant
struct ver_link {
  flck::atomic_write_once<long> time_stamp;
  ver_link* next_version;
  void* value;
  ver_link(long stamp,ver_link* next,void* value) // constructor
    : time_stamp(stamp), next_version(next), value(value) {} };
memory_pool<ver_link> links;
std::atomic<long> global_stamp;
void increment_global_stamp(long stamp) {
  if (global_stamp.load() == stamp)
     global_stamp.compare_exchange_strong(stamp,stamp+1);}
thread_local long local_stamp = -1;
```

Algorithm 5.4: Helper functions and types for indirect versioned pointers implementation in Algorithm 5.5.

all `std::atomic<T>` types (i.e. mutable shared locations holding values of type `T`) should be replaced with `flck::atomic<T>`.

Cost Bounds The `store` and `cas` operations each take a contant number of steps. [3] The `load` operation outside a `with_snapshot` takes a constant number of steps, and inside, the number of steps is at most proportional to the number of `store` and `cas` operations on the same versioned pointer that are concurrent with the containing `with_snapshot`. The overhead of the `with_snapshot` is a constant additive number of steps and, if using optTS, then the thunk f in a `with_snapshot`(f) might be run twice. We note that the number of steps does not tell the whole story since the time for a memory instruction, especially timestamp increments, can depend significantly on contention—hence the need to consider different time stamping mechanisms.

5.4 Versioning with Locks

We first describe an implementation of the interface that integrates locks, both blocking and lock-free, with the snapshot approach from Chapter 4. The code is given in Algorithms 5.4 and 5.5. The code in red is new to this chapter and the rest implements the snapshotting approach from Chapter 4. There are three important extensions in this section, the first two are specific to lock-free locks and the third is useful for either blocking or lock-free locks.

The first extension is in the implementation of `set_stamp` (Line 22), which is used by `load`, `store` and `cas` (three times) for set-stamp helping (i.e., changing a stamp from `tbd` to the current stamp). Usually with lock-free locks, code needs to be made idempotent by using `flck::atomic` for shared variables. Unfortunately making the global time stamp idempotent is expensive due to the high contention, which is amplified by helping threads. The key observation in our code is that setting the stamp does not need to be idempotent, and can use the non idempotent

[3]If used with Flocks lock-free locks and because of the way it avoids ABA with tagging, in the infrequent event that the tags run out, then the store and CAS can take longer.

```
template <typename F>
auto with_snapshot(F f) {
  local_stamp = global_stamp.load();
  increment_global_stamp(local_stamp);
  auto r = f();  local_stamp = -1; return r;}
template <typename V>
struct versioned_ptr {
  flck::atomic<ver_link*> v;
  static ver_link* set_stamp(ver_link* ptr) {
    if (ptr->time_stamp.load_non_idempotent() == tbd) {
      long n_t = global_stamp.load();
      if (ptr->time_stamp.load_non_idempotent() == tbd)
        ptr->time_stamp.cas_non_idempotent(tbd, n_t);
    } return ptr;}
  V* read_snapshot(long timestamp) {
    ver_link* head = set_stamp(v.load());
    while (head->time_stamp.load() > timestamp)
      head = head->next_version;
    return (V*) head->value; }
  bool cas_from_cam(ver_link* old_v, ver_link* new_v) {
    v.cam(old_v, new_v)
    return (v.load()==new_v || new_v->time_stamp.load() != tbd);}
public:
  versioned_ptr(V* ptr) : v(links.alloc(zero, nullptr, ptr}) {}
  ~versioned_ptr() { links.retire(v.load()); }
  V* load() {
    if (local_stamp != -1) return read_snapshot(local_stamp);
    else return (V*) set_stamp(v.load())->value;}
  void store(V* ptr) {
    ver_link* old_link = v.load();
    ver_link* new_link = links.alloc(tbd, old_link, ptr);
    v = new_link;
    set_stamp(new_link);
    links.retire(old_link);}
  bool cas(V* old_v, V* new_v) {
    ver_link* old_link = set_stamp(v.load());
    if (old_v != old_link->value) return false;
    if (old_v == new_v) return true;
    ver_link* new_link = links.alloc(tbd, old_link, new_v);
    if (cas_from_cam(old_link, new_link)) {
      set_stamp(new_link);
      links.retire(old_link);
      return true; }
    set_stamp(v.load());
    links.retire(new_link);
    return false; }
} // end versioned_ptr
```

Algorithm 5.5: Indirect versioned pointers in C++ safe for either blocking or lock-free locks.

versions of CAS and load. This is justified by the following theorem along with the fact than with lock-free locks helpers run in the same epoch as the original.

Theorem 5.4.1. *Any call to* set_stamp *on Line 22 of Algorithm 5.5 can be repeated any number of times by helper operations invoked after the original and with the same epoch without affecting the correctness of versioning.*

Proof (outline). Since the original and all helpers are in the same epoch, the pointer ptr must refer to the same logical object (i.e., what is pointed to has not been freed and reused). Therefore, if ptr->time_stamp is set from tbd to a valid timestamp, it will never change again. Some number of concurrent applications of set_stamp(ptr) could pass the first condition and run get_global_update_stamp. Depending on the implementation of stamps, this might increment the global timestamp multiple times. However, extra increments never change the external behavior. Furthermore, only the first cas_non_idempotent among the original and helping operations can succeed on Line 26 since the stamp can be set at most once. This will set the timestamp to a global stamp that existed between the invocation and response of the original set_stamp, or leave it as it was as a valid stamp if all fail. This is the condition required for correct set-stamp helping. □

The second extension is a cas_from_cam function (Line 33) needed when using a CAS while holding a lock-free lock. Recall that Flock does not supply an idempotent CAS since implementing one is quite difficult in general. Using a CAM followed by a load to check for success does not work since another CAM could succeed between the two operations making it appear that the first failed. It is possible to implement an idempotent CAS using a double-word wide regular CAS [18], but this is impractical since it would require that all versioned pointers be maintained as double words. Instead we take advantage of the existing indirection and timestamp to detect if the CAM was successful. cas_from_cam is idempotent because it uses idempotent load and CAM, so all that remains is to prove it simulates a CAS.

Theorem 5.4.2. *The* cas_from_cam *function on Line 33 and as used on Line 53 of Algorithm 5.5 implements a linearizable* CAS.

Proof. We first note that two concurrent cas_from_cams must have different new values since they both just allocated new objects on line 52. If the CAM on Line 34 failed then the first test on Line 35 will always fail since no concurrent cas_from_cam can be writing the same value. If the CAM succeeded but another CAM linearizes before the first check on Line 35 then this first check will fail. However, in this case the CAS responsible for the second CAM must have loaded the result of the first CAM into old_link on Line 49 to succeed on Line 34. In this case it must also have set the timestamp of old_link on Line 49. Therefore the second test on Line 35 will succeed and the cas_from_cam will properly return success. □

The third extension is to directly implement a store (line 42), which avoids several steps that would be required if a load and CAS are used to implement the store. An important assumption for the store is that it has no write-write races—i.e., that locks prevent two processes storing to the same location concurrently. We do not assume this for the CAS, and we allow read-write races. The store avoids checking and setting the timestamp on the old value (Lines 49), and avoids the need for a `cas_from_cam` (Line 53). Both of these simplifications are due to having no write-write races. Setting the stamp on the old value is not needed since the stamp must already be set by the previous completed write. The check if the `CAS` is successful is avoided since, without a race, the write must be the only successful write.

5.5 Indirection-on-need

In this section, we optimize the versioned pointer algorithm from the previous section to avoid indirection whenever possible. The idea is to check, on every store or CAS, if the pointer being written is to a newly allocated object that has not been written before. If so, we can avoid allocating an indirect `ver_link` by storing the `timestamp` and `next_version` fields directly in the newly allocated object. This requires adding two new fields to each object that might be referenced by a `versioned_ptr`, which is done through our library by inheriting from the `vp:versioned` class (e.g. Line 1 of Algorithm 3.1). Crucially, the decision of whether or not indirection is needed is made internally by our library and requires no additional steps by the user. Algorithm 5.6 shows how to support this optimization. We do not show the code for `CAS` but it follows a similar form.

The `timestamp` field of a new object is initialized to `tbd` to indicate it has not yet been referenced by any `versioned_ptr`. The first time a pointer to an object `O` is stored in a `versioned_ptr`, the necessary version list metadata is written directly into `O` (line 43), which also sets `O`'s timestamp (line 45). The next time this happens, the `store` will see that `O`'s timestamp has already been set (line 41), so it allocates a new `ver_link` to store the version list metadata, and this adds a level of indirection as before. We borrow a bit from each versioned pointer to distinguish between a direct versioned pointer and an indirect one.

To use the approach we make one reasonable restriction: when a versioned object is allocated by a process, a versioned pointer to it must be written using a store or `CAS` before any other process can see it—i.e., no side channels can be used to communicate the pointer. This is to avoid races among processes each trying to be the first to write a pointer to a newly allocated object. Using this restriction, we can prove the following Lemma.

Lemma 5.5.1. *Just before executing Line 44,* `new_v` *is only known to the current process and its* `timestamp` *is* `tbd`*.*

Proof. On entry to a `store`, if `ptr` is not a `nullptr` and its `timestamp` is `tbd`, then no other process has this pointer. This is because any previous store or `CAS` on `ptr` would have set the stamp, and by our restriction only the process that allocated the object has the pointer `ptr` to it before such a store or `CAS`. In this case just before executing Line 44 `new_v` is the same pointer as `ptr`, and it has not yet been communicated to any other process, so it satisfied the claimed property. If, on entry, `ptr` is either a `nullptr` or its `timestamp` is set (something other than `tbd`) then `new_v` is assigned a newly allocated version link (Line 42). Again the claimed property holds. □

Theorem 5.5.2. *In Algorithm 5.6 the* `store`*s properly linearize between their invocation and response and with respect to all loads (either inside a* `with_snapshot` *or not).*

```
struct versioned {
  flck::atomic_write_once<long> timestamp;
  ver_link* next_version;
  versioned(ver_link* next) : // constructor
    timestamp(tbd), next_version(next) {} };
struct ver_link : versioned {
  void* value;
  ver_link(ver_link* next, void* value) : // constructor
    versioned(next), value(value) {} };
  ...
template <typename V>
struct versioned_ptr {
  flck::atomic<ver_link*> v;
  ver_link* add_indirect(ver_link* ptr);    // adds/strips/tests bit of
  ver_link* strip_indirect(ver_link* ptr); // pointer to mark as indirect
  bool is_indirect(ver_link* ptr);
  V* read_snapshot(long timestamp) {
    ver_link* head = v.load();
    set_stamp(strip_indirect(head));
    while (strip_indirect(head)->timestamp.load() > timestamp)
      head = strip_indirect(head)->next_version;
    if (!is_indirect(head)) return (V*) head;
    else return (V*) strip_indirect(head)->value; }
  versioned_ptr(V* ptr) : v((ver_link*) ptr) {
    if (ptr != nullptr && ptr->timestamp.load() == tbd)
      ptr->timestamp = zero_stamp; }
  ~versioned_ptr() {
    ver_link* ptr = v.load();
    if (is_indirect(ptr)) links.retire(strip_indirect(ptr)); }
  ...
  V* load() {
    if (local_stamp != -1) return read_snapshot(local_stamp);
    else {
      ver_link* head = v.load();
      set_stamp(strip_indirect(head));
      if (!is_indirect(head)) return (V*) head;
      else (V*) strip_indirect(head)->value; }}
  void store(V* ptr) {
    ver_link* old_v = v.load();
    ver_link* new_v = (ver_link*) ptr;
    bool indirect = (ptr==null || ptr->timestamp.load() != tbd);
    if (indirect) new_v = add_indirect(links.alloc(old_v, ptr));
    else ptr->next_version = old_v;
    v = new_v;
    set_stamp(strip_indirect(new_v));
    if (is_indirect(old_v)) links.retire(strip_indirect(old_v));}
} \\ end versioned_ptr
```

Algorithm 5.6: Indirection-on-need optimization. Variables and functions unchanged from Algorithm 5.4 and 5.5 are omitted.

```
// ≤ global_stamp and all local_stamps of with_snapshots
long done_stamp;
template <typename V>
struct versioned_ptr {
  // call shortcut(head) before returning on line 36 of Alg. 5.6
  void shortcut(ver_link* ptr) {
    ver_link* ptr_ = strip_indirect(ptr)
    if (ptr_->timestamp.load_non_idempotent() <= done_stamp)
      if (v.cas_non_idempotent(ptr, (ver_link*) ptr_->value))
        links.retire_non_idempotent(ptr_); }
  void store(V* ptr) {
    // ... lines 39-43 from Alg. 5.6 here
    if(cas_from_cam(old_v, new_v)) {
      if(is_indirect(old_v)) links.retire(strip_indirect(old_v));
    } else v = new_v;
    set_stamp(new_v);
    if (indirect) shortcut(new_v); } } // end versioned_ptr
```

Algorithm 5.7: Shortcutting optimization. Variables and functions unchanged from Algorithm 5.6 are omitted.

Proof. (Outline) Lemma 5.5.1 implies that after the set_stamp on Line 45, the timestamp will hold a global timestamp (its linearization time) that falls between when the store on Line 44 happened, and the return of the set_stamp. This is because either the process set the stamp itself, picking a timestamp that falls within the claimed range or a helper did. If a helper did it must also have picked a stamp in the claimed range since it could not have see new_v before the store on Line 44, but must have picked and set the stamp before the process did. Finally any loads that come across the pointer new_v will run set_stamp on it forcing it to linearize before the load. Hence the store properly linearizes between its invocation and response. □

The advantage of indirection-on-need is that in many commonly used concurrent data structures [62, 80, 122], indirection is only potentially added when deleting a node since inserts always write newly allocated nodes. However, the indirect version links added by deletes eventually build up, and we need an efficient strategy for shortcutting them out.

Shortcutting. To identify indirect ver_links that can be safely shortcutted out, the idea is to make use of the memory reclamation scheme. Memory reclamation is essentially the problem of determining when an object or a version link is safe to garbage collect. If a versioned pointer is stored indirectly and all of the versions in its version list are safe to collect except the current one, then it is safe to shortcut out the version list by storing the versioned pointer directly.

In the following discussion we assume a shared done_stamp is maintained that is guaranteed at all times to be no greater than the minimum of the local_stamps of any ongoing with_snapshots as well as the global stamp. This ensures that no current or future read_snapshot will ask for a version older than done_stamp. At the end of the section, we describe how to maintain the done_stamp with epoch-based memory reclamation (EBR).

```
long cur_epoch_stamp;
std::atomic<long> global_epoch;
void update_epoch(long epoch) {
  long cur_stamp = global_stamp.load();
  if (global_epoch.compare_exchange_strong(epoch, epoch+1)) {
    done_stamp = cur_epoch_stamp;
    cur_epoch_stamp = cur_stamp; } }
```

Algorithm 5.8: Updating the `done_stamp` with EBR.

The code for shortcutting is given in Algorithm 5.7. Since all ongoing `with_snapshots` have timestamps no less than `done_stamp` (by assumption) we can determine if a version list is no longer needed by checking if the timestamp of the current version is no more than `done_stamp`. This check is performed by the `shortcut` function (Line 8). If the check passes, then no ongoing or future `with_snapshot` will access any of the old versions from this list, so it safely shortcuts out the version link (line 9). The `shortcut` function is called each time an indirect versioned pointer is `load`ed and also at the end of each `store` and `cas`. If there are no concurrent `with_snapshots`, then `store`/`cas` will immediately shortcut out any indirect nodes that it creates, in which case indirect nodes are only reachable for a brief moment of time. Shortcutting adds an additional write operation to each store/CAS, but we see in our experiments that the benefits almost always outweigh the cost. Note that `shortcut` uses non-idemptent versions of `load` and `cas` just like `set_stamp`. This still ensures idempotence at a higher-level because, even if a instance of `shortcut` is repeated multiple times by helpers, only the first to execute the `cas` on line 9 can succeed.

Shortcutting makes versioned pointer's `store` (and `CAS`) operation more difficult because a `store` needs to know if it overwrote an indirect pointer and is thus responsible for retiring it. Since shortcutting can be performed by concurrent processes at any moment, Algorithm 5.7 uses `cas_from_cam` to update `v` on line 9. A CAS could be used here if lock-free locks were not being used. If the `cas_from_cam` succeeds and `old_v` is indirect then `old_v` is retired (line 14). If it fails, then there was a concurrent shortcut operation changing `v` to a direct pointer, so we try updating `v` again, this time with a low-level store since it cannot be updated concurrently again (line 15). We sketch a proof of correctness below.

Theorem 5.5.3. *The shortcutting technique in Algorithm 5.7 maintains correctness of Algorithm 5.6.*

Proof. (outline) Shortcutting mostly affects `read_snapshot` operations, so this proof sketch will focus on showing that it does not change the return value of any `read_snapshot`. Suppose a versioned pointer is indirect (i.e. its `v` field points to a `ver_link` L), and `shortcut` later changes it to directly point to an object O. We argue that this change does not affect the return value of any ongoing or future `read_snapshot` operations. This shortcut causes any `read_snapshot` that would have visited L to instead visit O. Let S be a `read_snapshot` that would have visited L if not for the shortcut. We claim that the while loop in S will stop on both L and O because both L and O have timestamp at most S's timestamp. If S stops on L, then it will read and return L's `value` field which points to O. If S stops on O, then it will also return a pointer to O, so shortcutting does not change S's return value. So all that remains is to prove the claim that L

```
thread_local bool aborted, optimistic;
// add after Line 21 of Algorithm 5.6, in read_snapshot
if (strip_indirect(head)->time_stamp.load() == timestamp)
   aborted = optimistic;
template <typename F>
auto optimistic_with_snapshot(F f) {
  local_stamp = global_stamp;
  aborted = false; optimistic = true;
  auto r = f();   // run optimistically
  if (aborted) {  // rerun with incremented stamp if aborted
    aborted = false; optimistic = false;
    increment_global_stamp(local_stamp);
    r = f(); }
  local_stamp = -1; return r; }
```

Algorithm 5.9: Optimistic Timestamping.

and O have timestamp at most S's timestamp. At the moment when L is shortcut, its timestamp is at most `done_stamp`. We maintain that `done_stamp` is less than or equal to the timestamp of any ongoing or future `read_snapshot`, so it is less than or equal to S's timestamp. Therefore L's timestamp is at most S's timestamp by transitivity. Next we argue that O's timestamp is at most L's timestamp. This is because O's timestamp must be set before L, which is a `ver_link`, can point to it, and L's timestamp is set after it points to O, so L will be assigned a timestamp at least as high as the one in O. This proves the claim.

□

Maintaining the `done_stamp`. We now describe to how to maintain the `done_stamp` with epoch-based memory reclamation (EBR) [72]. Most multiversioning implementations [7, 123, 124, 171], including VERLIB and the versioned CAS approach from the previous chapter use EBR. EBR divides the execution into epochs with an epoch counter specifying the current epoch. This counter is separate from the timestamp counter. EBR ensures that all active operations started during either the current or the previous epoch, and therefore any objects removed during earlier epochs are safe to reclaim.

Using EBR to maintain `done_stamp` requires modifying the code for incrementing the global epoch counter so that it records the timestamp at the start of each epoch. These modifications are shown in Algorithm 5.8, and the global variables `cur_epoch_stamp` and `done_stamp` (from Algorithm 5.7) store the timestamp at the start of the current and previous epochs, respectively. With EBR, `update_epoch` is never called concurrently with different epoch numbers as arguments, so there is never a race to update `cur_epoch_stamp` or `done_stamp`.

5.6 Optimistic Timestamps

A shared global timestamp is used by `with_snapshot` and update operations to get and increment the current time. In Algorithm 5.5 the queries (`with_snapshot`) increment the timestamp, as in Chapter 4 and other works [7, 67]. It is also possible to instead have the updates increment the stamps [123], or use a high-frequency hardware clock that is synchronized across cores [96]. We call these implementations QUERYTS, UPDATETS, and HWTS, respectively. HWTS is not portable across machines, and QUERYTS and UPDATETS increment the timestamp too frequently in certain workloads resulting in poor performance due to contention on the stamp.

In the context of software transactional memory, it has been observed that transactions can run optimistically, without incrementing the timestamp, and only increment the timestamp if the optimistic execution aborts [55]. In this section, we describe a simplified version of the optimistic timestamping optimization presented in TL2 [55] specialized for our setting of read-only transactions. In our experimental evaluation, we refer to this as OPTTS. Using this approach, a query only needs to abort if it comes across a timestamp equal to its own. The approach runs the query at most twice since the second run will not abort.

The rather simple code for the approach is given in Algorithm 5.9. It uses the `global_stamp` and `increment_global_stamp` defined in Algorithm 5.5. The approach modifies `read_snapshot` so after locating the version with the largest timestamp less than or equal to the current local stamp, it checks if that stamp is equal to the current stamp (Line 3). If so, and if running optimistically, it sets the abort flag. The approach then modifies `with_snapshot(f)` so it first runs the query `f` without incrementing the stamp (Line 9). It then checks if the query aborted and, if so, increments the stamp and reruns (Line 13). The second run is guaranteed to produce a linearizable return value because it is essentially the same as the old `with_snapshot` implementation in Algorithm 5.5. Note that this technique requires `f` to be safe to run twice. This is a natural requirement since `f` is a read-only query on the data structure. The following theorem gives linearization points for this new optimistic implementation of `with_snapshot`.

Theorem 5.6.1. *If* `with_snapshot(f)` *runs* `f` *without an abort, then it is linearized when it reads the global stamp (Line 7). Otherwise, it is linearized when it increments the global stamp (Line 12).*

Proof. In the first case, all pointer versions encountered by `f` either have strictly larger or strictly smaller timestamp than `f`. The strictly larger ones were added after the linearization point of `f`, so `read_snapshot` in Algorithm 5.6 correctly skips them to look for older versions. The strictly smaller ones were added before `f`, and `read_snapshot` returns the latest pointer version with a smaller timestamp. Therefore, `read_snapshot` returns the version of each pointer at the linearization point of `f`.

In the second case, we can ignore the optimistic execution and the non-optimistic execution performs the same steps as `with_snapshot` implemented with QUERYTS in Algorithm 5.5. The linearization point is correct because it is the same as the linearization point for `with_snapshot` in Algorithm 5.5. □

OPTTS works well in range query heavy workloads because most of the queries can proceed optimistically. It also works well in update heavy workloads because updates never increment the global timestamp. As an optimization, queries passed to `with_snapshot` can check the abort flag and finish early if they see it set.

5.7 Experimental Evaluation

We apply VERLIB to several concurrent set data structures to add support for linearizable range queries and groups of *k* find operations that act atomically (multi-finds). Our goal is to (1) measure the overhead VERLIB adds to the original data structure, (2) compare the optimization levels and timestamp implementations described in Sections 5.5 and 5.6, respectively, and (3) compare with state-of-the-art concurrent set data structures. We plan to make VERLIB, as well as the code for these experiments, publicly available.

Setup. Our experiments ran on a 64-core Amazon Web Service c6i-metal instance with 2x Intel(R) Xeon(R) Platinum 8375C (32 cores, 2.9GHz and 108MB L3 cache), and 256GB memory. Each core is 2-way hyperthreaded, giving 128 hyperthreads. We used `numactl -i all`, evenly spreading the memory pages across the sockets in a round-robin fashion. The machine runs Ubuntu 22.04.1 LTS. The C++ code was compiled with g++ 11 with `-O3`. Jemalloc was used for scalable memory allocation. For Java, we used OpenJDK 19.0.1 with flags `-server`, `-Xms50G` and `-Xmx50G`. We report the average of 3 runs, each of 5 seconds. For Java we also pre-ran 3 runs to warm up the JVM.

Data Structures. In the section, we report on five data structures implemented in C++ with VERLIB: a b-tree, an adaptive radix tree (`arttree`), a doubly-linked list (`dlist`), a singly-linked list (`list`), and a `hashtable`. For the first four we used existing data structures from the FLOCK library [22] and applied the modifications described in Section 5.3. These four are lock based and can be run in either blocking or lock-free mode using FLOCK or VERLIB. The doubly-linked list code is given in Section 5.1.1. The hash table is a `CAS` based implementation that maintains an array per bucket, and copies the array on update [48]. We tune the number of buckets to maintain a 1/2 load factor. For all five data structures, we implemented range queries (which search for all integers within the range) and multi-finds, wrapping them in a `with_snapshot`.

All of the original data structures required recording more than once. However, the only place the b-tree recorded a node more than once was at the root, so we created a strictly recorded-once version to compare to. We also implemented a recorded-once version of the singly linked list—on deleting a node it copies the next node. We did not create recorded once implementations of the other structures.

We compare these VERLIB data structures with several state-of-the-art concurrent set data structures: `LFCA` [170], Jiffy [96], EpochBST [7], BundledSkiplist [123], BundledCitrus [123], LSKN-arttree [103, 104], SB-abtree [151]. The first three are lock-free and all except the last two support linearizable range queries. LSKN-arttree is a concurrent radix tree and the others are comparison based ordered set data structures. We used implementations by the original authors for all these data structures. We use the C++ version of most data structures to be consistent with our implementations. For LFCA and Jiffy, we were unable to find a reliable C++ implementation and had to use the Java implementation instead.

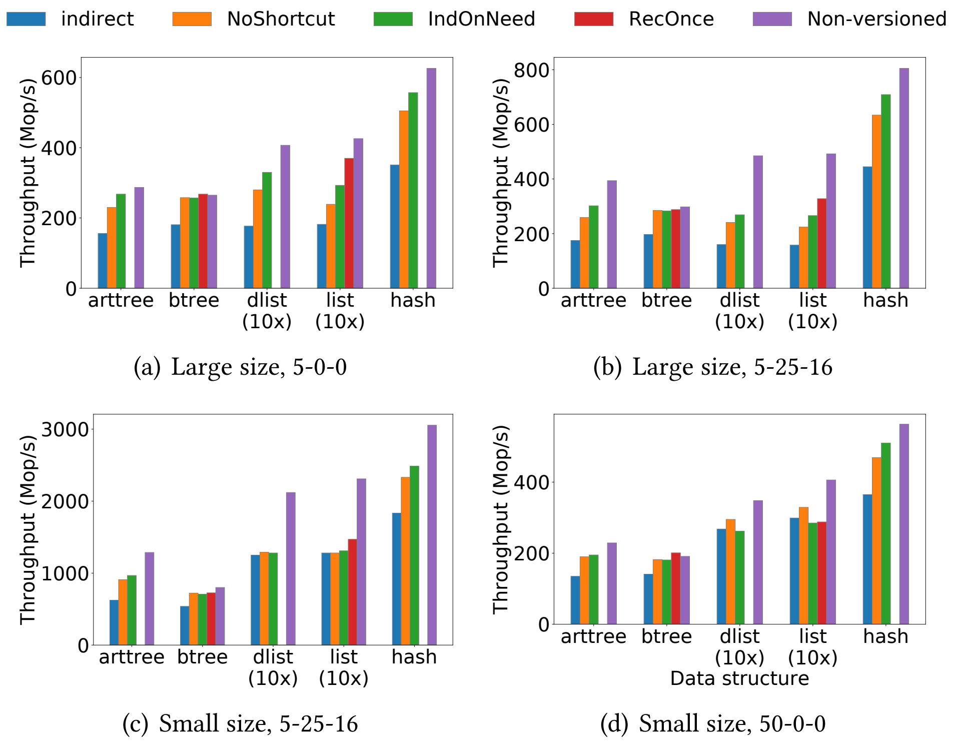


Figure 5.10: Comparing different versioned pointer implementations. 5-25-16 to denotes a workload where each thread performs 5% updates and 25% multi-finds of size 16. Keys are drawn from an uniform distribution and each run uses 128 threads. List (10x) indicates that its throughput was scaled up by a factor of 10 to make the graphs more readable.

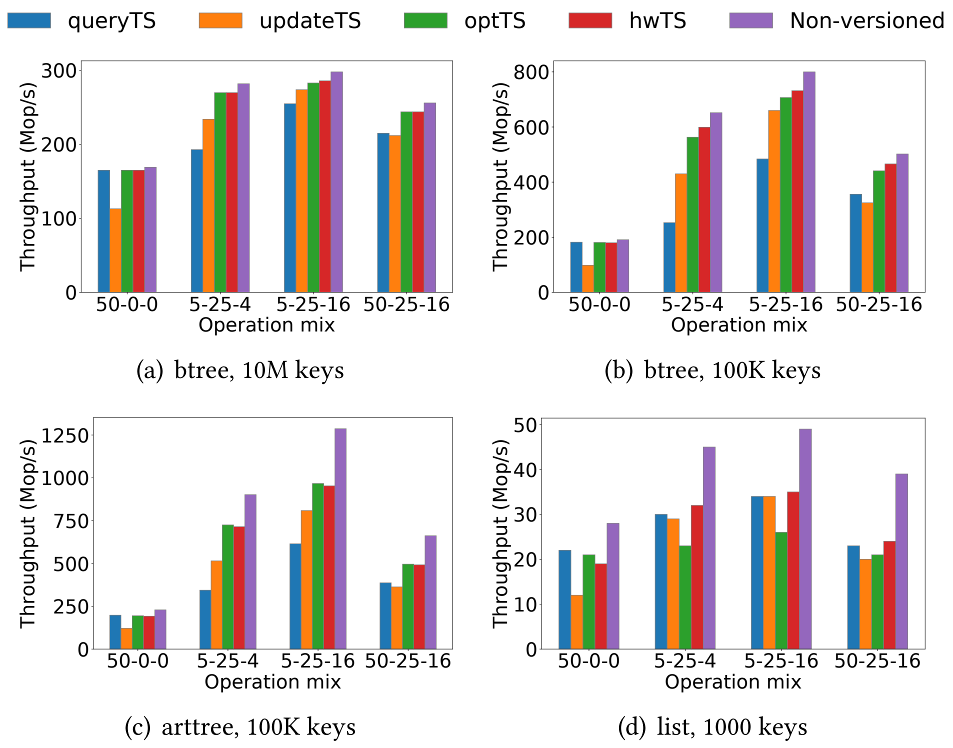


Figure 5.11: Comparing different timestamp implementations. 5-25-16 denotes a workload where each thread performs 5% updates and 25% multi-finds of size 16. Keys are drawn from an uniform distribution and each run uses 128 threads.

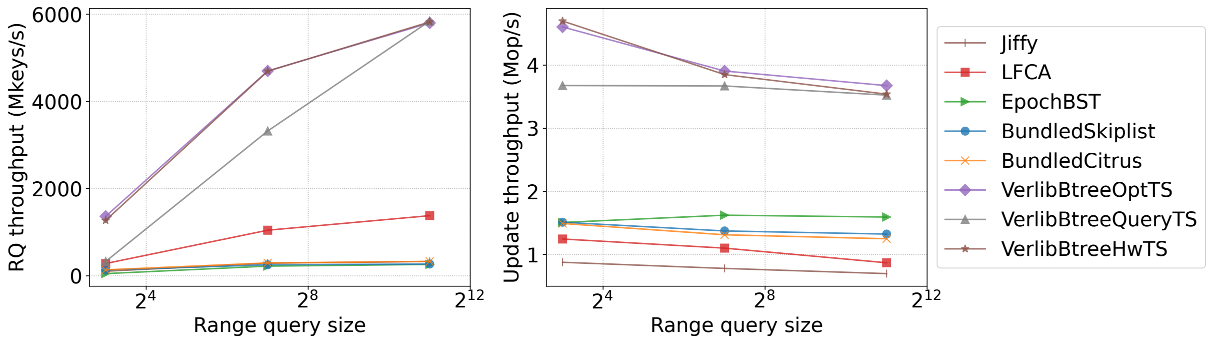
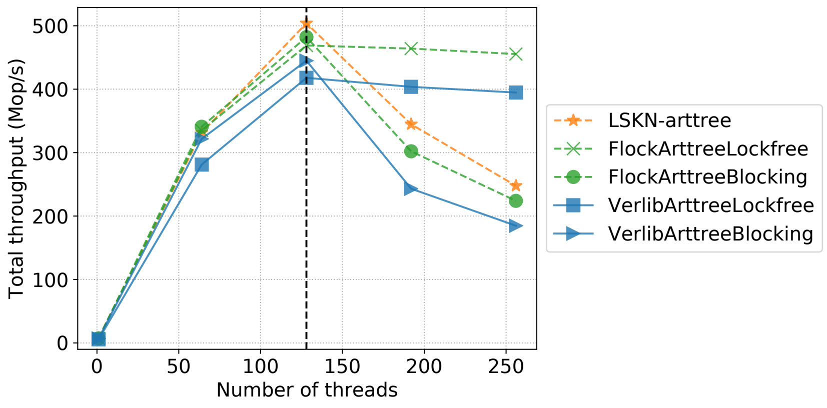
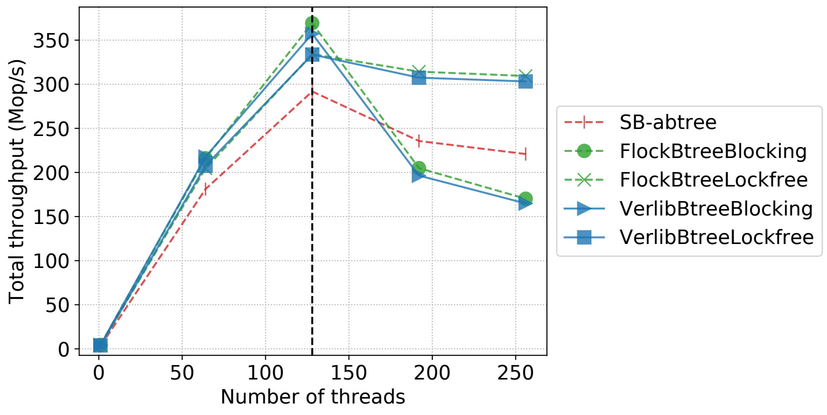


Figure 5.12: Comparing various data structures supporting linearizable range queries. Run with 100 threads: 5 update threads, 95 range query threads, keys drawn from uniform distribution, 10M data structure size



(a) arttree



(b) btree

Figure 5.13: Comparing various arttree and btree implementations. Solid lines used for data structures that support linearizable range queries, dotted lines used otherwise. Run with 10M keys, 5% updates, 95% lookups, and keys drawn from Zipfian distribution. The dotted vertical line indicates the number of cores on our machine.

Workloads. In our experiments, we vary the following parameters: (a) data structure size (denoted by n), (b) operation mix, (c) size of range queries/multi-finds (denoted by s), (d) number of threads, and (e) the distribution from which keys are drawn. In most experiments, we initialize each data structure with either $n = 100K$ or $n = 10M$ keys by running a mix of inserts and deletes on an initially empty data structure. These sizes are chosen to illustrate performance when the data set fits and does not fit into the L3 cache. For linked lists, we instead use $n = 100$ and $n = 1000$ as the two sizes. In the timed portion of the code, each thread performs a mix of operations, consisting of inserts and deletes (done in equal numbers), as well as finds and either range queries or multi-finds. We use a universe U of $2n$ distinct 64-bit keys chosen uniformly at random. Keys for all operations (including initialization) are drawn randomly from U, which ensures that the size of the data structure remains approximately n throughout the experiment. Keys are drawn using either the uniform distribution or Zipfian distribution with parameter 0.99, which is the default in the YCSB benchmark [43]. Our range queries search for all keys in the range $[a, b]$ where a is drawn from U as before and b is chosen so that the expected number of keys in the range is s.

5.7.1 Results

Indirection on need. Figure 5.10 compares the performance of the versioned pointer algorithms presented in this chapter. Specifically, `Indirect` represents the algorithm from Section 5.4, `NoShortcut` uses indirection-on-need but without shortcutting (Algorithm 5.6), and `IndOnNeed` also uses shortcutting (Algorithm 5.7) and is the default implementation in VERLIB. We also implemented a variant of versioned pointers, called `RecOnce`, which never uses indirect nodes and only works for recorded-once data structures (as with the experiments in Chapter 4). We applied this to our recorded once variants of b-tree and `list`. All these variants use the optimistic timestamp technique (OPTTS) presented in Section 5.6. To measure the overhead of using versioned pointers, we also show the original non-versioned data structure (`Non-versioned`) in the graphs. Multi-finds on this data structure are not linearizable (each find can linearize at its own point).

Figures 5.10(a) and 5.10(b) are run on large data structures with 1000 keys for linked lists and 10M keys for the others. Figure 5.10(c) and 5.10(d) are run on small data structures with 100 keys for linked lists and 100K keys for the others. These are all run with uniform distribution—the Zipf distribution follow the same the trends. Overall, when indirection-on-need is used, the overhead of applying versioned pointers to a `Non-versioned` data structure is generally low.

For `arttree` and `hashtable`, indirection-on-need improves performance by 35%-72% relative to `Indirect` versioned pointer. We see that the shortcutting optimization also consistently helps on these data structures. However on small linked lists with lots of updates (Figure 5.10(d)), shortcutting sometimes hurts performance due to the extra stores. In this case, indirection-on-need without shortcutting is actually the fastest versioned pointer implementation.

For b-tree, `IndOnNeed` versioned pointers achieves essentially the same performance as `RecOnce`, while not requiring the data structure to be recorded-once. The same is true for `list` in Figure 5.10(c). In Figure 5.10(b), `list` with `RecOnce` is slightly faster than `IndOnNeed`, but Figure 5.10(d) shows that modifying `list` to be recorded-once also comes with some overhead as it requires locking additional nodes.

The remaining experiments use the `IndOnNeed` implementation of versioned pointers.

Timestamps. Figure 5.11 compares four different timestamp implementations: QUERYTS, UPDATETS, HWTS, and OPTTS. We observed the same trends for both uniform and Zipfian distribution and focus on uniform distribution in Figure 5.11.

Across these experiments, HWTS tends to perform the fastest because our machine supports a very light-weight `rdtsc` instruction for reading the hardware clock. Not all machines support a fast, synchronized hardware clock, so this implementation is not portable. Optimistic timestamp (OPTTS) achieves almost the same performance as HWTS in Figures 5.11(a)- 5.11(c), indicating that optimistic executions of multi-find often succeed without having to increment the global timestamp. However, it is up to 22% slower for long linked lists in Figure 5.11(d). This is because all updates to a linked list are on the same path, so it is more likely for an optimistic multi-find to encounter one of these updates and have to restart. On long lists, this restarting can be expensive. For linked lists of size 100 and also in most other cases, OPTTS is faster than QUERYTS and UPDATETS while being more portable than HWTS. QUERYTS and UPDATETS perform poorly in multi-point query heavy and update heavy workloads, respectively, due to high contention when incrementing the timestamp. This can cause them to be 2x slower than OPTTS (see Figure 5.11(b)).

Direct Stores. Section 5.4 described how to replace a load-then-`CAS` with a store, avoiding some checks and updates. We ran experiments with and without this optimization. On workloads with 50% updates we saw up to a 8% improvement in performance (e.g., on b-trees with 100K keys and uniform distribution). On loads with 5% updates the improvement was negligible, as might be expected since the optimization only affects the performance of updates.

Range query. Figure 5.12 compares our versioned btrees with state-of-the-art data structures supporting linearizable range queries. Updates and especially range queries on our versioned b-trees are significantly faster because of the increased cache locality due to the large fanout at internal nodes and the batching of keys in each leaf. Out of the other range queriable data structures, only `LFCA` stores a batch of keys in each leaf, however internal nodes still only have fanout 2. Developing a general and easy-to-apply library allowed us to apply versioning to faster baseline data structures than those used in previous work.

Scalability. Figure 5.13 measures the scalability of our versioned `arttree` and b-tree up to oversubscription. The previous experiments were run with VERLIB in lock-free mode, and these graphs also show its performance in blocking mode. Consistent with previous experiments on lock-free locks from Chapter 3, blocking mode tends to be slightly faster before oversubscription, but drops severely in performance after oversubscription. This motivates the importance of supporting both persistence and lock-free locks.

We also plot the performance of LSKN-arttree and SB-abtree, which is a state-of-the-art concurrent radix tree and B-tree, respectively. They use blocking locks, so they also slow down after oversubscription. Our VERLIB `arttrees` and b-trees perform competitively with these data structures while also being lock-free and supporting linearizable range queries.

5.8 Conclusion

In conclusion, this chapter presents an efficient implementation of concurrent versioned pointers that is compatible with both blocking and lock-free locks and is optimized to avoid indirection whenever possible. It is significantly easier to apply than versioned CAS objects from Chapter 4, which requires the user to often modify their data structure in non-trivial ways to get good performance. We also present an optimistic timestamping technique and show that it performs better than the commonly used increment on query and increment on update approaches.

We wrap these ideas in a VERLIB library and apply it to several data structures to support linearizable range queries. Experiments show that data structures are significantly faster than existing concurrent, range queriable data structures.

Related Work

Snapshot. Implementing a snapshot object is a classic problem in shared-memory computing with a long history. Ellen surveyed some of this work [70]. A *partial snapshot* object allows operations that take a snapshot of selected entries of the array instead of the whole array [10, 89]. An f*-array* [91] is another generalization of snapshot objects that allows a query operation that returns the value of a function f applied to a snapshot of the array. As mentioned above, snapshot objects have a less flexible interface than our approach to snapshotting.

We describe in Section 4.4 how to use our snapshots to support multi-point queries on a wide variety of data structures. Previous work has focused on supporting such queries on *specific* data structures. Bronson *et al.* [33] gave a blocking implementation of AVL trees that supports a `scan` operation that returns the state of the entire data structure. Prokopec *et al.* [138] gave a `scan` operation for a hash trie by making the trie persistent: updates copy the entire branch of nodes that they traverse. `Scan` operations have also been implemented for non-blocking queues [128, 129, 137] and deques [66]. Kallimanis and Kanellou [94] gave a dynamic graph data structure that allows atomic dynamic traversals of a path.

Range query. Range queries, which return all keys within a given range, have been studied for various implementations of ordered sets. Brown and Avni [36] gave an obstruction-free range query for k-ary search trees. Avni, Shavit and Suissa [11] described how to support range queries on skip lists. Basin *et al.* [15] described a concurrent implementation of a key-value map that supports range queries. Like our approach, it uses multi-versioning controlled by a global counter.

Fatourou, Papavasileiou and Ruppert [67] gave a persistent implementation of a binary search tree with wait-free range queries, also based on version lists. Our work borrows some of these ideas, but avoids the cumbersome handshaking and helping mechanism they use to synchronize between scan and update operations. This more streamlined approach makes our approach easier to generalize to other data structures. Winblad, Sagonas and Jonsson [170] also gave a concurrent binary search tree that supports range queries.

Some researchers have also taken steps towards the design of general techniques for supporting multi-point queries that can be applied to classes of data structures, although none are as general as our approach.

Petrank and Timnat [134] described how to add a non-blocking `scan` operation to non-blocking data structures such as linked lists and skip lists that implement a set abstract data type; `scan` returns the state of the entire data structure. Updates and scan operations must coordinate carefully using auxiliary *snap collector* objects. Agarwal *et al.* [3] discussed what properties a data structure must have in order for this technique to be applied. Chatterjee [42] adapted Petrank and Timnat's algorithm to support range queries.

Arbel-Raviv and Brown [7] described how to implement range queries for concurrent set data structures that use epoch-based memory reclamation. They assume there exists a traversal algorithm that is guaranteed to visit every item in the given range that is present in the data structure for the entire lifetime of the traversal.

Concurrently with our work, Nelson-Slivon, Hassan and Palmiery [123] describe a technique for supporting range queries on a variety of ordered data structures (e.g. linked list, skip list and binary search tree). Kobus and Kokociński, and Wojciechowski describe a linked-list data structure that supports arbitrary snapshots well as atomic batch updates [96] . Sheffi, Ramalhete and Petrank avoid long version lists in snapshots by aborting long-lived queries that force the system to hold onto to many queries [150]. All these use version lists and the last one is based on the technique presented in Section 4.2.

Multiversioning. Within the database and software transactional memory (STM) literature there has been a long history of having transactions capture a snapshot of the state using multi-versioning [19, 25, 40, 53, 57, 69, 98, 124, 131, 132, 133, 136, 144, 145, 153, 171]. This avoids conflicts between read-only transactions and write transactions. Indeed, the idea of version lists for this purpose dates back to Reed's thesis on transactions [144] and is implemented in many modern-day database systems. Much of the work, especially the earlier work, is lock-based. Fernandes and Cachopo [69] introduced a lock-free approach to transactional multiversioning. Their approach, however, fully sequentializes transactions that require updates by adding each successful transaction to the end of a transactional log. Other work has, for example, studied how to make updates in the past [57] by splicing elements into the version lists.

Multiversioning using version lists dates back to the 70s [144] and is commonly used for efficiently supporting read-only transactions in databases [25, 40, 53, 57, 69, 98, 107, 124, 131, 132, 133, 136, 144, 145, 171]. None of this work considers making concurrent data structures persistent, and only one [69] is lock-free and it fully sequentializes commits.

Timestamps have been used for multiversioning at least since the 70s [144]. It was quickly noticed that incrementing a global timestamp can be a bottleneck and there have been many attempts at reducing this bottleneck [25, 107, 158, 172]. Bernstein and Goodman suggest removing a global timestamp by using Lamport clocks [101] (written objects are given a stamp such that the partial ordering of the objects is consistent with any dependences). The approach ensures serializability but not strict serializability (i.e. linearizability). In particular read-only transaction can return results from any previous time (including the beginning of time). TicToc [172] uses a similar approach but not in the context of multiversioning. Other systems suggest using loosely synchronous clocks [107], or epoch-based clocks [158], but they also do not support strict serializability. Ruan, Lu and Spear [146] and Kobus and Kokociński, and Wojciechowski [96] suggest using synchronous hardware clocks available in some modern architectures. The exact guarantees of these clocks, however, is not well documented—perhaps vendors do not want to guarantee that they will supply fast synchronous clocks in all future platforms.

Part III

Safe Memory Reclamation

Introduction

Safe Memory Reclamation is the problem of freeing allocated memory that is shared across multiple threads. It is essential in any program that dynamically allocates memory. Both ease-of-use and efficiency are critical for safe memory reclamation, as concurrent data structures are often presented in the literature without describing how to recycle memory. It is often up to practitioners to add this step themselves. In this part, we present several new concurrent reference counting algorithms, which we wrap in an easy-to-use interface and release as an efficient C++ library.

Chapter 6

Deferred Concurrent Reference Counting

6.1 Introduction

Safe memory reclamation techniques can be broadly divided into two categories, manual and automatic. With manual techniques, the user is responsible for freeing objects. To protect against read-reclaim races, this is often performed with a *retire* operation, which defers the reclamation until it is safe, i.e., until no other thread is reading that object. Such techniques include read-copy-update (RCU) [79], epochs [71], hazard pointers [117], pass-the-buck [87], interval-based reclamation [167], hazard eras [142], and others [39].

Automatic techniques are similar to what can be found in garbage collectors, but without the ability to scan processor private root sets (registers, stacks, etc.). A common technique is reference counting [51, 87, 102, 115, 135, 154, 160], which consists of attaching a counter to each managed object that counts the number of pointers to it, and performing reclamation when the counter hits zero. Both manual and automatic techniques can be implemented as library interfaces, and both need to take care of read-reclaim races. Both can also have some advantages over garbage collectors, such as having more control over memory layout or guaranteeing lock-freedom.

In the context of concurrent data structures, manual techniques are often difficult to use and can lead to subtle and hard to reproduce bugs. As evidence, we note that the use of manual memory reclamation in several recent papers is incorrect (see Chapter 6.8 for more details). These errors can lead to memory leaks or even memory faults. Since these data structures and their use of memory reclamation were implemented and adopted by experts in the field, it would be difficult for common users to get them right.

Reference counting, on the other hand, requires very few modifications for programmers to integrate into their code, and provides memory safety and leak freedom automatically as long as the programmer either does not create reference cycles or breaks such cycles before they become unreachable. Owing to their ease of use, there has been an increase in interest in atomic reference-counted pointers, as evidenced by their inclusion in the most recent C++ standard (C++20). There also exists optimized open-source [64] and even commercial implementations [169]. However, for many concurrent data structures, reference counting can be expensive in practice due to the need to frequently increment and decrement shared counters [82].

A crucial challenge when designing a concurrent reference-counting scheme is dealing with read-reclaim races when the reference count reaches zero. In particular, if one thread decrements the counter to zero, initiating reclamation, at the same time that another thread increments the counter, the object will appear with a non-zero reference count, even though it is no longer safe to access. Various techniques have been developed to overcome this, including using tools from manual SMR to delay reclamation until there can no longer be any active reads [87, 154], and the split reference count technique [168, Chapter 7.2.4], which involves maintaining one internal reference count on the managed object itself and possibly several external reference counts, one on each pointer to the object.

In this chapter, we propose an efficient approach to automatic memory reclamation based on a novel combination of reference counting and manual SMR. We make several advances to make library-based concurrent reference counting both theoretically efficient and more practical. Theoretically, we show the first solution with constant expected time overhead using only single word compare-and-swap (CAS) and only delaying $O(P^2)$ decrements. Previous approaches are either only lock-free [46, 51, 87, 115, 160, 168], wait-free with $O(P)$ time [154] per operation, or use double-word fetch-and-add [102, 135], which is not available on modern machines.

Our approach is based on a new algorithm that generalizes hazard pointers to allow for multiple retires on the same object. Standard hazard pointers would not be efficient with multiple retires, requiring potentially much more space. This technique allows us to implement *deferred decrements* that protect an object's reference count, delaying decrements (and hence reclamation) while an increment is in progress. This contrasts with previous reference counting techniques [75, 87] that use hazard pointers to delay memory reclamation *after* a reference count reaches zero. This is a subtle difference, but it has ramifications both in theory and practice. This generalization of hazard pointers, which we refer to as *acquire-retire*, could be of interest beyond reference counting. We further extend the approach by borrowing the idea of *deferred increments* from reference-counted garbage collectors [12, 13, 26, 52, 105]. When a reference to an object is short lived, it almost certainly doesn't need to modify the reference count. We can facilitate this using acquire-retire to protect the reference count during the reference's short lifetime. In the common case, this avoids both the increment and the decrement. Putting both these ideas together, we call the resulting algorithm CDRC which stands for concurrent deferred reference counting.

We have implemented our technique as a library for C++[1] and show that it is more efficient than existing optimized libraries for atomic reference-counted pointers [46, 64, 169]. Our experiments show that deferred decrements alone lead to improved performance over classic approaches, and that deferred increments can result in a substantial speedup–over an order of magnitude on highly contended workloads.

Lastly, we show that our scheme performs well against state-of-the-art manual SMR techniques from a recent benchmark suite [125, 167]. When applied to a range of concurrent data structures for which reference counting previously achieved no scaling whatsoever, our technique keeps up and scales alongside the fastest manual SMR techniques. Furthermore, it manages to achieve throughput rates within a factor of 1.2-2.5x of the fastest manual SMR techniques that

[1]Available at https://github.com/cmuparlay/concurrent_deferred_rc

use an unbounded amount of memory while consuming only a modest amount itself. Its memory consumption and performance is competitive with hazard pointers but, unlike hazard pointers, it is not constrained to a limited class of data structures. Last but not least, our scheme is automatic and hence easier and safer to use than manual ones. To summarize, the contributions of this chapter are:

- A generalization of hazard pointers that supports constant-time acquire and allows multiple concurrent retires of the same handle, which we call *acquire-retire*,
- the design of a theoretically efficient scheme for automatic memory reclamation based on combining acquire-retire and reference counting,
- a practical implementation of the technique as a library for C++, evaluated on a comprehensive set of benchmarks which show that it outperforms existing reference-counting techniques, and is also competitive with manual SMR.

6.2 Related Work

Manual SMR Manual SMR techniques can be broadly classified as either protected-pointer-based or protected-region-based.

Protected-pointer-based methods. These methods work by identifying specific objects or memory locations that are currently in use and hence should not be destroyed/freed. The collection part of the algorithm is responsible for ensuring that it never frees something that is currently in use. Hazard-pointers [117] is one of the most widely used protected-pointer-based techniques. The main idea is that every process has some globally visible array of "hazard pointers". When a process wishes to read a mutable shared pointer, it *announces* its intention to do so by writing the pointer into one of the hazard pointers. This may require a retry if the value of the pointer changes before the announcement is complete. When the process has finished reading or manipulating the shared object, it *releases* the hazard pointer by clearing the announcement. When a process removes a node from the data structure and wishes to free it, it instead *retires* the node, which places it in a *retired list* of nodes pending deletion. A process that wishes to reclaim memory must scan the hazard array of every process to ensure that it does not reclaim anything currently announced. Nodes in the retired list that are not announced are safe to free.

Several variants of hazard pointers exist, many of them designed to help implement other memory reclamation schemes. Herlihy *et al.* [87] develop Pass The Buck (PTB), which is used to implement their algorithm for lock-free reference counting. Correia *et al.* [46] develop pass-the-pointer (PTP), which improves on the memory bounds of traditional hazard pointers and is used to implement their own lock-free reference counting algorithm, OrcGC.

Protected-region-based methods Rather than protecting specific objects/memory locations, protected-region-based methods protect groups of objects. This generally results in lower synchronization cost (fewer memory barriers) and hence higher throughput, but at the cost of wasting more memory, since many objects will be protected even when they do not need to be. Epoch-based reclamation (EBR) [71] and Read-copy-update (RCU) [79] are the most widely used protected-region-based techniques. In EBR, the algorithm maintains a global timestamp called the epoch. Whenever a memory location is retired, it is placed in a retired list corresponding to the current epoch. When the user wishes to begin an operation that will access or modify shared state, the executing thread announces the value of the current epoch. When every thread has announced the value of the current epoch, the retired list from the previous epoch can be freed and the epoch can advance to the next value. Note that this is safe because if an object is retired at epoch e and every process has subsequently announced epoch $e + 1$, then any thread that was performing an operation at the time of the retire has since completed. DEBRA [39] is an optimized implementation of EBR with better practical performance.

Hazard Eras (HE) [125, 142] is a combination of protected-pointer- and protected-region-based methods. In HE, acquired pointers do not announce the pointer itself, but rather the epoch on which it was read. If the epoch changes infrequently, this results in fewer memory barriers than a full-blown protected-pointer-based scheme. In HE and Interval-based Reclamation (IBR) [167], each object is tagged with a birth epoch when it is allocated and a retire epoch when it is retired. In IBR, a retired object is safe to reclaim when no announced epoch intersects its birth-retire interval.

Hyaline [127] is a protected region approach that tags each retired object with a counter corresponding to the number of currently active operations. When an operation completes, it can decrement one from every object that retired during its operation interval. The operation that brings a counter to zero is responsible for freeing it. Crystalline [126] extends Hyaline with wait-freedom.

Atomic Reference Counting Lock-free reference counting (LFRC) was first described by Detlefs *et al.* [51], but their algorithm requires a DCAS operation (a CAS on two independent words), which is not supported by any current architecture. Herlihy *et al.* [87] use their PTB technique to obtain an algorithm for single-word lock-free reference counting (SLFRC). The idea is to use PTB to protect the reference count of the object from being freed while a process is attempting to increment it. Sundell [154] developed the first wait-free algorithm for reference counting, however, some of their operations cost $O(P)$ time.

In the practical world, the C++ standard recently added support for *atomic shared pointers* [113], which provide a thread-safe way for multiprocessor environments to share reference-counted pointers. Prior implementations of atomic operations on shared pointers use a small global hash table of locks, and hence are not scalable in practice. We know of two external libraries that support lock-free solutions [64, 169]. Both are based on the split reference count technique [168, Chapter 7.2.4] and are lock-free, but not wait-free. Prior to this, a similar technique was developed by Lee [102] and generalized by Plyukhin [135]. Their version is constant time but requires atomic double-word fetch-and-add on a location containing both a pointer and an unbounded sequence number. Unlike double-word CAS, double-word fetch-and-add is not supported by modern machine architectures.

The split reference count technique [168] is a lock-free solution for atomic reference counting. It involves splitting the reference count into an internal count, and an external count on each mutable shared reference. Loads from shared references increment the corresponding external count, while local releases decrement the internal count instead. When a shared reference is discarded, its accumulated external count minus one is added to the internal count. While this technique does not rely on SMR, it tends to scale poorly in practice since loads must be performed with a double-word CAS to increment the external count.

The major performance drawback of reference counting is the necessity to increment the reference count each time an object is read. Recent work has addressed this by developing solutions for reference counting that allow safe reads without incrementing the reference count. Tripp *et al.* [157] implement Fast Reference Counter (FRC). FRC uses deferred reference counting and a per-thread root set (equivalent to an announcement array of hazard pointers) to achieve low contention and enable safe reads of managed objects without incrementing the reference count. Correia *et al.* [46] develop OrcGC, which uses their PTP technique to implement reference-counted pointers that can also be safely read without incrementing. There has also been orthogonal work on reducing the amount of contention caused by increments and decrements to the same reference count [1, 61]. The idea is to implement a *Scalable NonZero Indicator*, or SNZI, object which is a relaxed counter that only indicates whether or not the counter is non-zero. Taking advantage of this relaxation, increment, decrement and query operations on SNZI objects often need not contend with each other.

Deferred Reference Counting. Deutsch and Bobrow [52] introduce *deferred reference counting* for garbage collectors, which consists in eagerly counting references present in the heap, but ignoring those in registers and on the stack. Objects that reach a heap reference count of zero are placed in a "zero-count table". Periodically, the garbage collector then scans the stack and registers to determine which objects in the zero-count table are reachable, removing them from the zero-count table, or which are unreachable, and hence can be safely destroyed. Subsequent work by Bacon et al. [12], Levanoni and Petrank [105], and Blackburn and McKinley [26] further build on the idea of deferred reference counting, identifying additional situations in which reference-count updates can be deferred or elided entirely.

Unlike our method, all of this prior work focuses specifically on languages with automatic garbage collection and require pausing processes and hence are not lock-free. Although we borrow the name "deferred reference counting“ due to the high-level conceptual similarities, our techniques and methods are substantially different because they apply to manually memory-managed languages.

6.3 Overview of Our Approach

Recall that the difficulty of implementing safe concurrent reference counting is the possibility for a race between a decrement that sets the count to zero, initiating reclamation, and an increment, which increments the counter back above zero, giving the appearance that the managed resource is still live. Our idea is, intuitively, that if there is an increment racing with a decrement, to delay the decrement until after the increment has completed.

Our key insight is that this can be achieved by applying a hazard-pointers-like scheme where the resource being protected is neither a memory block nor a managed object, but rather the reference count itself that is attached to a managed object. This leads to a simple algorithm for concurrent reference counting. To obtain a new pointer to a reference-counted object, our algorithm *acquires* the reference count of the object to protect it, then increments the counter and *releases* the protection. To discard a pointer, the reference count of the object to which it points is *retired*, which, when ejected (i.e., at some point in time when the reference count is not acquired by any increment), decrements the reference count, deleting the managed object and reclaiming the memory if it reaches zero. By delaying decrements until all the increments that started before it complete, we ensure that an object is safe to collect as soon as its reference count reaches zero. This contrasts with previous techniques [75, 87] that perform decrements eagerly and use SMR to delay memory reclamation *after* a reference count reaches zero.

Note that in our algorithm, a reference count could be retired multiple times before being ejected a single time. This could happen, for example, in an execution where three pointers to the same reference-counted object are discarded concurrently. Traditional hazard pointers interfaces [86, 117, 120] explicitly disallow resources from being retired more than once, which make sense in the SMR setting, but not when managing more general resources such as reference counts. To support these more diverse use cases, we define a generalization of hazard pointers called *acquire-retire* and show how to implement it efficiently, with all operations taking only constant time in expectation.

Lastly, we extend our reference-counting algorithm, which defers decrements, with what we call *private pointers*, which can be thought of as *deferring increments.* When a reference to an object is short lived, such as during the traversal of a linked data structure, a standard reference-counting scheme would have to increment and decrement the reference count in quick succession. Instead, we observe that we can apply acquire-retire to temporarily protect the reference count during the private pointer's lifetime. This avoids both the increment and the decrement, which we show substantially improves the practical performance of our scheme.

6.3.1 Our Reference-Counting Library

To illustrate and evaluate our techniques, we implemented them as a library for C++. Our implementation makes use of standards-compliant C++ features, including C++11 atomics and memory orderings, and uses no OS- or architecture-dependent code. In this section, we briefly discuss the interface of our library, compare it to the interfaces of other memory reclamation techniques, and discuss an important practical feature that allow us to efficiently implement a range of concurrent data structures.

Our library consists of three class templates, starting with `atomic_shared_ptr<T>`, which provides thread-safe management of a `shared_ptr<T>`, which manages a reference-counted pointer to an object of type `T`. The `atomic_shared_ptr<T>` interface is modelled after `atomic<shared_ptr<T>>` in the C++ standard, while `shared_ptr<T>` is designed to closely mimic `shared_ptr<T>`. Lastly, we provide `private_ptr<T>`, which facilitates low-cost reads of an object managed by an `atomic_shared_ptr<T>` by protecting it with a deferred increment, rather than an explicit increment of the reference counter. We describe the usage of these types in more detail in the following sections.

atomic_shared_ptr. `atomic_shared_ptr<T>` is closely modelled after C++'s `atomic<shared_ptr<T>>`. It provides support for all of the standard operations, such as atomic load, store, and CAS.

- **load**`()`. Atomically creates a `shared_ptr` to the currently managed object, returning the `shared_ptr`.

- **load_private**(). Atomically creates a private_ptr to the currently managed object, returning the private_ptr.
- **store**(desired). Atomically replaces the currently managed pointer with desired, which may be either a shared_ptr or a private_ptr.
- **compare_and_swap**(expected, desired). Atomically compares the managed pointer with expected, and if they are equal, replaces the managed pointer with desired. The types of expected and desired may be either shared_ptr or private_ptr, and need not be the same.
- **compare_exchange_weak**(expected, desired). Same as compare_and_swap but, if the managed pointer is not equal to expected, loads the currently managed pointer into expected. This operation may spuriously return false, i.e. it is possible that the value of expected does not change.

The most interesting point of the interface is that it supports two flavors of load operations, **load** and **load_private**, which return shared_ptr and private_ptr respectively.

shared_ptr and private_ptr. The shared_ptr type is closely modelled after C++'s standard library shared_ptr. It supports all pointer-like operations, such as dereferencing, i.e. obtaining a reference to the underlying managed object, and assignment of another shared_ptr to replace the current one. It is safe to read/copy a shared_ptr concurrently from many threads, as long as there is never a race between one thread updating the shared_ptr and another reading it. Such a situation should be handled by an atomic_shared_ptr.

The private_ptr type supports all of the same operations as shared_ptr. The only differences between the two is that while shared_ptr can safely be shared between threads, private_ptr can only be used locally by the thread that created it and cannot be copied. The use of private_ptr should result in better performance than shared_ptr provided that each thread does not hold too many private_ptr at once. If a thread exceeds the soft limit on private_ptr (see Section 6.5.2), their performance will degrade to similar to or slightly worse than shared_ptr. Therefore, private_ptr should be used for reading typically short-lived local references, for example, reading nodes in a data structure while traversing it.

To illustrate our library and the three types, we refer to an implementation of a concurrent stack in Figure 6.1, which we elaborate on in the next section. The head node of the stack is stored in an atomic_shared_ptr because it may be modified and read concurrently by multiple threads. Each node of the stack stores its next pointer as a non-atomic shared_ptr. This is safe, because although multiple threads may read the same pointer concurrently, the internal nodes of the stack are never modified, only the head is. Lastly, we can use a private_ptr while performing pop_front, since reading the head is a short-lived local reference that will never be shared with another thread.

Support for Marked Pointers. A common optimization in concurrent data structures is to steal some of the unused bits from a pointer to mark links in the data structure as pending deletion. Since our reference-counted pointer algorithm uses plain single-word pointers and does not internally steal any bits, it is possible to expose those redundant bits to the programmer for them to use in this fashion. Our pointer types therefore include a customization point that allows a markable pointer type to be used in place of raw pointers internally, and allows custom behavior to be added via a policy class. We have used this to implement *markable* versions of our types that offer `get_mark`, `set_mark`, and `compare_and_set_mark`, which require no manual bit twiddling from the programmer, allowing them to easily and efficiently implement data structures with marked links.

6.3.2 Usability Comparison to Manual SMR

Our interface is closely modeled after and designed to be as easy to use as the standard C++ types. In Figures 6.1- 6.3, we depict three implementations of a concurrent stack, using our library, hazard pointers, and RCU. Our code avoids the potential pitfalls of manual SMR, as it is impossible to read the value stored in `head` without protecting it automatically, and no manual retires are necessary. Although calling retire is quite simple in this example, it is not always so easy. Figure 6.4 depicts a snippet of code from an implementation of the Natarajan and Mittal tree [122]. This code cleans up deleted nodes from the tree by swinging a pointer from a node to one of its descendants. It is a subtle but important detail to notice that in the presence of concurrent updates, this operation may delete multiple nodes, and hence may be required to retire many nodes, not just a single one. In Section 6.8, we discuss how this bug and others have appeared in the artifacts of several published papers written by memory management experts.

6.4 Defining the Acquire-Retire Interface

We propose a generalization of hazard pointers for resource management called *acquire-retire*. As with hazard pointers, it supports four operations: acquire, release, retire, and eject. The generalization is that it allows multiple retires of the same handle, which is critical in our reference-counting implementation. An *acquire* takes a pointer to a location containing a resource handle, reads the handle and *protects* the resource, returning the handle. A later paired *release*, releases the protection. A *retire* is used to indicate the resource is no longer needed. A later paired *eject* will return the resource handle indicating it is no longer protected and safe to destruct. We say a retire, or its corresponding destruct, is *delayed* between the retire and when its handle is ejected. For our time and space bounds, we require that every retire is followed by at least one eject. All operations are linearizable [88], i.e., must appear to be atomic.

We describe a constant-time implementation of acquire-retire, that only requires single-word memory instructions, and for P processors and K protected resources has $O(PK)$ memory overhead. Describing an efficient implementation of acquire-retire requires two insights. The first is that hazard pointers can be combined with a recent result on atomic copy [28] to ensure constant-time acquire. The second insight is that multiple concurrent retires of the same handle can be supported by appropriately keeping track of multiplicity.

Allowing multiple retires of the same handle makes defining the behavior of **retire** and **eject** more subtle. The high-level approach is to associate **acquire** operations with **retire** operations rather than handles. In our interface, **acquire**(ptr, *ann*) takes as input a pointer to a memory location (ptr) storing a resource handle, and a pointer to an announcement slot (*ann*). It returns the handle stored at the memory location. The **release**(*ann*) operation takes as input an announcement slot, and has no return value. In a sequential execution, we say that an **acquire**(ptr, *ann*) operation is *active* between its execution and the execution of either the next

```
struct Node { T t; shared_ptr<Node> next; }
atomic_shared_ptr<Node> head;

void push_front(T t) {
  shared_ptr<Node> p = make_rc<Node>(t, head.load());
  while (!head.compare_exchange_weak(p->next, p)) {}
}

optional<T> pop_front() {
  private_ptr<Node> p = head.load_private();
  while (p != nullptr && !head.compare_exchange_weak(p, p->next)) {}
  if (p != nullptr) return {p->t};
  else return {};
}

```

Figure 6.1: C++ implementations of an ABA-safe, concurrent stack using our library.

```
struct Node : rcu_obj_base<Node> { T t; Node* next; };
atomic<Node*> head;

void push_front(T t) {
  auto p = new Node{{}, t, head.load()};
  while (!head.compare_exchange_weak(p->next, p)) {}
}

optional<T> pop_front() {
  rcu_reader guard;
  auto p = head.load();
  while (p != nullptr && !head.compare_exchange_weak(p, p->next)) {}
  if (p != nullptr) {
    p->retire();
    return {p->t};
  }
  else return {};
}

```

Figure 6.2: C++ implementations of an ABA-safe, concurrent stack using RCU. The syntax for RCU is based on a C++ standards proposal [119].

```
struct Node : hazptr_obj_base<Node> { T t; Node* next; };
atomic<Node*> head;

void push_front(T t) {
  auto p = new Node{{}, t, head.load()};
  while (!head.compare_exchange_weak(p->next, p)) {}
}

optional<T> pop_front() {
  Node* p;
  hazptr_holder h;
  do {
    p = h.get_protected(head);
    if (p == nullptr) return {};
  } while (!head.compare_exchange_weak(p, p->next)) { }
  if (p != nullptr) {
    p->retire();
    return {p->t};
  }
  else return {};
}
```

Figure 6.3: C++ implementations of an ABA-safe, concurrent stack using hazard pointers. The syntax for hazard pointers based on a C++ standards proposal [119], and is implemented in Folly [64].

```
void cleanup() {
  ...
  /* Update the left child of ancestor to point to sibling */
  if(ancestor.left->compare_and_swap(successor, sibling)) {
    /* retire nodes on path from successor to sibling */
    for(Node* n = successor; n != subling;) {
      Node* tmp = n;
      if(getFlag(n->left)) {
        retire(n->left);
        n = n->right;
      } else {
        retire(n->right);
        n = n->left;
      }
      retire(tmp);
    }
    return true;
  } else return false; }
```

Figure 6.4: Manually calling retire is easy to forget and it sometimes adds non-trivial code. The highlighted portion of the code is not needed in our library.

acquire($*$, *ann*) operation or the next **release**(*ann*) operation, whichever comes first. After this point, the **acquire** is said to be *inactive*. The **retire**(h) operation takes as input a handle and the **eject** operation either returns $\perp$ or a handle.

Our implementation requires that **acquire**/**release** operations on the same announcement slot are never concurrent with each other. Typically, each process will have its own private set of announcement slots. Announcement slots can either be allocated statically, or dynamically as threads are created and retired, in the same way as hazard pointers [117]. We formally specify the behaviour of the interface below.

Definition 4 (Acquire-Retire)**.** *Any proper, concurrent execution can be linearized to a sequential history with the following guarantees:*

1. *Each* **acquire***(ptr, $*$) returns the handle currently stored in the memory location pointed to by ptr.*
2. *Let f be a function that maps each* **acquire** *returning h to either a later* **retire***(h) or $\perp$. Let g be an injective (one-to-one) function that maps each* **eject** *returning h to an earlier* **retire***(h). For all f, there is a g such that whenever $f(A) = g(E)$, the* **acquire** *A is inactive by the time* **eject** *E is executed.*

We note that Definition 4 captures our intuition of what the interface is supposed to protect against. In particular, it ensures that any destruct of a resource placed after the **retire** and **eject** will happen after all processes **release** that resource. If there are multiple retires on the same handle, it ensures that each is mapped to at most one **eject**.

Definition 4 never forces **eject** operations to return a handle, so for an implementation of acquire-retire to be useful, it has to provide some guarantees on how often **retire**s are ejected. We say a **retire** is ejected if there is an **eject** mapped to the **retire**. Assuming each call to **retire** is always followed by a call to **eject**, our algorithm ensures that there are always no more than $O(KP)$ **retire**s that have not been ejected, where K is the total number of announcement slots. We defer the description of our algorithm for acquire-retire until Section 6.6.

6.5 Deferred Reference Counting

Armed with the acquire-retire technique, we now describe our algorithms for reference counting with deferred decrements and increments. The interface supports atomically storing to, loading from, and CASing into a mutable reference-counted pointer in a shared location. Our algorithms support these operations with constant-time overhead, have $O(P^2)$ memory overhead, and defer at most $O(P^2)$ reference-count decrements (see Theorem 1). We note that deferred increments is just a practical optimization which does not affect these bounds.

Both algorithms also have the useful property that references are implemented as raw pointers, which means two things. First, that a reference occupies just a single word, unlike some implementations [169] which use a double-word representation and require a double-word CAS. Second, that we do not "steal" any bits of the pointer representation, as is done by some libraries [64]. This is important in some applications, since it leaves unused bits of the pointer representation for the user to utilize, which is necessary in many common implementations of lock-free data structures that "mark" pointers. For example, the Harris linked list [80] or Natarajan and Mittal's binary search tree [122].

6.5.1 Deferred Decrements

Recall that the race we are trying to avoid when designing a scheme for concurrent reference counting occurs when one thread removes a reference, decrementing the corresponding counter to zero, at the same time that another thread creates a new reference, incrementing the counter. Such races in which a location can be simultaneously read by one thread and updated by another can occur in just about any lock-free data structure. Our approach solves this problem by using acquire-retire to protect the reference count and defer decrements from being applied while there is a potential increment in progress. We say a decrement is *deferred* if a reference has been overwritten or otherwise deleted, but the count on the corresponding managed object has not yet been decremented. The eject operation on the reference count corresponds to decrementing the count and, if it goes to zero, reclaiming the managed object.

Algorithms and Analysis. Figure 6.5 depicts our algorithm using a reference-counting interface similar to the one used by Herlihy et al. [87] and Detlefs et al. [51]. We assume each reference-counted object has a counter attached that can be atomically incremented or decremented with **addCounter**, which returns the old value.

The **load** operation atomically loads a pointer from a shared memory location into a local pointer and returns it. Since **load** creates a new reference to the object, it increments the reference count. To protect against a potential race between this increment and a decrement setting the count to zero, the increment is surrounded by an **acquire** and **release**.

The **store** operation atomically copies a local pointer into a shared memory location. Since this creates an additional reference to `desired`, it first increments the reference count. Note the subtle detail that unlike in **load**, this increment does not need to be protected by an **acquire** and **release**. This is because the existence of the argument `desired` guarantees that the reference count is at least one, and hence cannot race to zero during this operation. Our implementation writes into the shared memory location using a fetch-and-store operation so that it can decrement the reference count of the pointer that was overwritten. Decrementing the reference count immediately would introduce a race, so instead, we defer the decrement by retiring the pointer. Each **retire** is always followed by an **eject** of a previously retired pointer, which is then decremented. Recall that pairing each **retire** with an **eject** is what allows acquire-retire to yield efficient time and space bounds. Notice that a process might have the same pointer in its retired list multiple times, which is why we need the more general acquire-retire interface rather than hazard pointers.

The **cas** operation works similarly to **store**, except that it only modifies the reference counts if the underlying CAS succeeds. Note that for safety reasons, **cas** must first protect `desired` with an **acquire** before performing the CAS. If it did not, the CAS could succeed right before another thread stored to `A`, which could cause the reference count of `desired` to be decremented. If this decrement took the count to zero, initiating reclamation, the object would be unsafely destroyed before the **cas** had a chance to increment the reference count.

```
using ref = Object*;

AnnouncementSlot announcement[P];

ref load(ref* A) {
  ref ptr = acquire(A, &announcement[pid]);
  if (ptr != nullptr) increment(ptr);
  release(&announcement[pid]);
  return ptr; }

void store(ref* A, ref desired) {
  if (desired != nullptr) increment(desired);
  ref current = fetch_and_store(A, desired);
  if (current != nullptr) {
    retire_and_eject(current); }

bool cas(ref* A, ref expected, ref desired) {
  ref ptr = acquire(&desired, &announcement[pid]);
  if (compare_and_swap(A, expected, desired)) {
    if (desired != nullptr) increment(desired);
    if (expected != nullptr) {
      retire_and_eject(expected); }
    release(&announcement[pid]);
    return true;
  } else {
    release(&announcement[pid]);
    return false; } }

void destruct(ref ptr){
  if (ptr != nullptr) {
    decrement(ptr); } }

void retire_and_eject(ref ptr) {
  retire(ptr);
  optional<|ref|> e = eject();
  if (e != ⊥) decrement(e); }

void increment(ref ptr) {
  ptr->addCounter(1); }

void decrement(ref ptr){
  if (ptr->addCounter(-1) == 1) {
    delete ptr; } }
```

Figure 6.5: Operations for atomic reference-counted pointers with deferred decrements. pid is the unique id of the current processor, $0 \leq \texttt{pid} < P$.

The **destruct** operation takes as input a reference-counted pointer that is no longer needed and destroys it. In an object-oriented language, such as C++ or Rust, this would be handled automatically by the reference-counted pointer's destructor. Note that since it would be an unsafe race to read from a pointer while it was being destroyed, **destruct** does not have to call **retire** but can instead eagerly decrement.

The **retire_and_eject**, **increment** and **decrement** operations are used internally and are not part of the interface. The **decrement** operation is responsible for initiating reclamation if the counter is decremented to zero. In our pseudocode, by **delete**, we mean to destroy the underlying `Object`, which includes recursively calling **destruct** on any reference-counted pointers it owns, and reclaiming the memory it occupies.

Result 1 (Deferred Reference Counting): *On P processes, any number of reference-counted objects with references stored in shared mutable locations supporting atomic load, store, and CAS can be implemented safely with:*

1. *references as just pointers (i.e., single-word addresses),*
2. *$O(1)$ time for load,*
3. *$O(1)$ expected time for store and CAS excluding the cost of any call to* **delete** *resulting from a* **decrement**
4. *$O(P^2)$ space overhead and $O(P^2)$ deferred decrements,*
5. *only single-word read, write, CAS, fetch-and-store, and fetch-and-add.*

This implies constant-time overhead since the deletion of the retired objects is required by any non-trivial reclamation scheme.

Proof. We first consider safety. The key property we need to prove is that between Lines 6 and 8 of a **load**, the reference count of **ptr** never hits 0 and therefore the object pointed to by **ptr** never gets collected while its reference count is being incremented.

To apply Definition 4 of the acquire-retire interface, we first define a function f which specifies a mapping from **acquire**s to **retire**s (or $\perp$). Consider an **acquire** performed by Line 6 of **load** operation L. This **acquire** was run on a shared memory location and suppose it returns the pointer ptr. If there is a subsequent **store** or **cas** on this memory location that overwrites ptr and calls **retire** on ptr, then f maps the **acquire** to this **retire**. Otherwise, f maps the **acquire** to $\perp$. Suppose ptr gets overwritten by a subsequent **store** or **cas** operation S. The decrement of ptr caused by S will be delayed until after some later **eject** gets mapped to the **retire** in S. By Item 2 of Definition 4, an **eject** cannot be mapped to this **retire** until the **acquire** from L is **release**d, so the reference count of **ptr** is at least 1 as long as the **acquire** is active.

We now consider the four properties from Result 1. (1) References are just pointers, as claimed. The $O(1)$ expected time for reading and overwriting references (2) and $O(P^2)$ space (3) follow directly from the acquire-retire results. The number of delayed decrements is at most $O(P^2)$ (3) because there are at most $O(P^2)$ delayed retires. The implementation uses the primitives used by acquire-retire and a FAA for incrementing and decrementing the reference count (4). □

Copy versus Move Semantics. Our algorithms in Figure 6.5 implement **store** and **cas** with *copy semantics*. That is, since they effectively create a new reference to `desired`, they increment the corresponding reference count. In many practical situations however, the caller may have no subsequent use for their copy of `desired`, which may be soon to be destructed, leading to a decrement of the reference count. In this situation, it is favorable to implement versions of **store** and **cas** that have *move semantics*, i.e., that *consume* the copy of `desired` passed as an argument. This removes the need to increment the reference count since the caller gives up their count. Our C++ library implements this optimization.

6.5.2 Deferred Increments / Private Pointers

A big performance bottleneck that appears when implementing concurrent data structures using pure reference counting occurs when traversing linked nodes. On a node-based concurrent data structure, a safe traversal requires temporarily incrementing and decrementing the reference counts of all the nodes encountered to prevent them from being deleted while being read. This is inefficient for multiple reasons; increments and decrements must be performed with an atomic fetch-and-add instruction, and these may contend if multiple processors are operating on the same node concurrently.

This contrasts with other SMR techniques such as hazard pointers, which just perform a write to a single-writer location for each node traversed, or epoch-based methods, which perform a single write before beginning the traversal. Neither of these methods experience any contention. Furthermore, due to cache coherency protocols, incrementing the reference count of a node reserves the cache line in exclusive mode, causing the other processes to experience a cache miss the next time they access this node.

In the previous section, we gave algorithms for reference counting with deferred decrements. Although they achieve constant-time overhead, they are still prone to the practical performance hit of frequent increments. To improve our scheme in practice, we therefore introduce the notion of *deferred increments*. Specifically, if an algorithm needs to briefly protect an object, such as during the traversal of a linked data structure, but does not need to keep a long-lasting reference, we observe that there is no need to eagerly increment the reference count. Instead, the algorithm can use the existing infrastructure of acquire-retire to temporarily prevent any decrements from being applied while the reference is held. The downside is that a pointer protected in this manner can only be accessed by the thread that created it, so we call it a *private pointer*. Private pointers prevent deferred decrements from being applied while they are held, and when they are released the protection can be cleared, resulting in no change to the reference counter.

By using reference counting to protect long-lived references, such as links inside the data structure, and private pointers to protect short-lived references, we obtain the best of both worlds – the ability to traverse the data structure without introducing contention, without the burden of having to manually retire nodes that are no longer reachable. This is not possible with a pure reference counting or pure SMR (e.g., hazard pointers) approach.

Private Pointer Implementation. We show the implementation for private pointers in Figure 6.6. The `load_private` operation is similar to `load`, except that it returns a `private_ptr`, which is a protected local reference coupled with an `AnnouncementSlot`. When a private pointer

```
using private_ptr = pair<|ref, AnnouncementSlot*|>;
const int MAX_PRIVATE_PTRS_PER_THREAD = 7;

AnnouncementSlot announce[P][MAX_PRIVATE_PTRS_PER_THREAD];
thread_local int next;

private_ptr load_private(ref* A) {
  AnnouncementSlot* slot = get_slot()
  ref ptr = acquire(A, slot);
  return {ptr, slot}; }

void release_private(private_ptr S) {
  auto [ptr, slot] = S;
  if (ptr != nullptr) {
    if (slot->read() == ptr) release(slot);
    else decrement(ptr); } }

AnnouncementSlot* get_slot() {
  for (int i = 0; i < MAX_PRIVATE_PTRS_PER_THREAD; i++)
    if (announce[pid][i].read() == ⊥)
      return &announce[pid][i];
  AnnouncementSlot* slot = &announce[pid][next];
  increment(slot->read())
  next = (next + 1) % MAX_PRIVATE_PTRS_PER_THREAD;
  return slot; }

void destruct(ref ptr){
  if (ptr != nullptr) {
    retire_and_eject(ptr); } }
```

Figure 6.6: Interface and algorithm for private pointers. This algorithm is compatible with the reference-counting algorithm of Figure 6.5, except that the **destruct** operation from Figure 6.5 must be replaced with the one given here.

is no longer needed, it can be released with the `release_private` method. Note that the same process that acquired the private pointer must release it.

Since multiple private pointers may need to be held by a single processor, our implementation allocates seven additional announcement slots per processor. This means that the eight total announcement slots of a process fit on a single cache line on common architectures. By packing them into a single cache line, the **ejectAll** method of acquire-retire does not suffer any noticeable performance loss.

When a process wishes to acquire a private pointer, the algorithm scans its announcement slots and selects the first empty slot it finds. If no slots are available, it selects one of the existing slots and eagerly increments the reference count on the protected object (i.e., it applies the deferred increment) and takes over the slot for itself. In our implementation, the slot to take over is selected in a round-robin fashion. When a private pointer is released, it checks whether its announcement slot has been reused, and if so, correspondingly decrements the reference count. Otherwise, no decrement is necessary, and the announcement can simply be released.

Lastly, to safely hold private pointers, we need to slightly modify the `destruct` operation for references. If a private pointer is loaded from a shared reference, and that reference is subsequently updated, the reference count cannot be eagerly decremented, or the object protected by the private pointer might be destroyed. Instead, the decrement must be deferred by calling `retire`.

6.6 Acquire-Retire Algorithm

We now describe how to implement constant-time **acquire**, **release**, **retire**, and expected constant-time **eject**. This algorithm uses techniques from hazard pointers [117] and pass-the-buck[87] with some changes to support the more general acquire-retire interface.

The standard lock-free version of **acquire** from hazard pointers executes a loop in which the pointer to be protected is read from a shared location and written into a local announcement slot. Each iteration, the pointer is re-read from the shared location to check whether it still matches the one that was announced. To reduce the complexity of **acquire** to constant time, we leverage a recently proposed primitive called `swcopy` [28], which atomically copies from one location to another location, but requires that the destination location is only written to by a single process. Note that making the read of the shared location and write to the announcement slot appear to happen atomically is precisely the purpose of the lock-free acquire loop, and hence, by replacing it with a `swcopy`, we can implement **acquire** in constant time. Blelloch and Wei [28] present an implementation of M `Destination` objects using $O(M + P^2)$ space such that `read`, `write` and `swcopy` all take constant time.

A **release**(`ann`) operation unprotects by simply clearing the announcement slot `ann`, and **retire**(`x`) simply adds `x` to a process-local retired list called `rlist`. To determine which handles are safe to eject, the **ejectAll**(`rl`) method loops through all the announcement slots and makes a list of all the handles that it sees. We call this list of handles `plist` for "protected list". If a handle is seen multiple times in A, then it will also appear that many times in `plist` (this differs from standard hazard arrays). Next, **ejectAll** computes a multi-set difference between `rl` and `plist`. This step can be implemented in $O(|rl| + K)$ expected time using a local hash table, where K is

```
using AnnouncementSlot = Destination<|optional<|T|>|>;

thread_local list<|T|> rlist;
thread_local list<|T|> flist;

T acquire(T* ptr, AnnouncementSlot* ann) {
  ann->swcopy(ptr);
  return ann->read(); }

void release(AnnouncementSlot* ann) {
  ann->write(⊥); }

void retire(T t) {
  rlist.add(t); }

optional<|T|> eject() {
  perform steps towards ejectAll(rlist);
  if (!flist.is_empty())
    return flist.pop();
  return ⊥; }

void ejectAll(list<|T|> rl) {
  list<|T|> plist = empty;
  // loop through all existing AnnouncementSlots
  for each AnnouncementSlot* ann {
    optional<|T|> a = ann->read();
    if(a != ⊥) plist.add(a); }
  list<|T|> freed = multiSetDiff(rl, plist);
  flist.add(freed);
  rlist.remove(freed); }
```

Figure 6.7: Implementing acquire-retire. `Destination` is a destination object supporting atomic copies [28]. `slots` is a list of all of the announcement slots owned by all processors.

the total number of announcement slots. The result of this multi-set difference are handles that can be safely ejected without violating the specifications of acquire-retire. It is important that we keep track of multiplicity and perform multi-set difference because when a handle is retired multiple times, each occurrence of this handle in the announcement slots might be associated with a different **retire**. So if a handle appears in the retired list s times and the announcement slots t times, it is safe to eject only $s - t$ copies of this handle.

An **eject** is essentially a deamortized version of **ejectAll**. Every time it is called, it performs a small constant number of steps towards **ejectAll**(`rlist`), where each hash table operation counts as a single step. Thus **eject** takes expected constant time. When **ejectAll** returns a list of handles, they get removed from `rlist` and added to a local free list to be returned one at a time by the following **eject**s.

Pseudocode for this implementation appears in Figure 6.7 and its properties are summarized in Theorem 2.

Result 2 (Acquire-Retire): *For an arbitrary number of resources and locations, P processes, and at most K resources protected at any given time, the acquire-retire interface can be supported with:*

1. $O(1)$ *time for acquire, release, and retire,*
2. $O(1)$ *expected time for eject,*
3. $O(KP)$ *deferred retires,*
4. $O(KP)$ *space overhead assuming* $K \geq P$, *and*
5. *only single-word read, write and CAS*

Proof of Result 2. To show that Figure 6.7 is a linearizable implementation of the acquire-retire interface, we need to prove both properties in Definition 4. The first property says that an **acquire**(ptr, k) A returns the current value of $*ptr$ at the linearization point of A. This is easily ensured by linearizing A at the atomic copy of $*ptr$. Before moving on to the second property, we first define linearization points for the remaining three operations and introduce some useful notation. A **release** operation is linearized at the write instruction that clears the announcement slot and a **retire**(h) operation by process p is linearized when h is added to p's local retired list ($rlist$). An **eject** returning h is linearized when h is removed from the process's local free list ($flist$). We use **eject**(h) to denote an **eject** operation returning h and we use subscripts to indicate the process that performed a particular operation. For example **retire**$_p$ denotes a **retire** operation by process p. We extend our definition of active **acquire**s to apply to concurrent histories by saying that an **acquire**(ptr, k) operation is *active* between its linearization point and the linearization point of either the next **acquire**($*$, k) operation or the next **release**(k) operation, whichever comes first. After this point, the **acquire** is said to be *inactive*.

Now we show that our algorithm satisfies the second property of Definition 4. Let f be any function that maps each **acquire** returning h to either a later **retire**(h) (in linearization order) or $\perp$ We describe how to construct an injective function g from each **eject** returning h to an earlier **retire**(h) (in linearization order) such that whenever $f(A) = g(E)$, the **acquire** A is inactive at the linearization point of the **eject** E. To construct g, whenever a handle is added to a process's local retired list, we logically tag the handle with the **retire** operation that added it. Given a execution, for every write to the announcement array by an **acquire** operation A, we logically tag the write with the retire operation $f(A)$, if $f(A) \neq \perp$. Whenever a handle gets returned by **eject**, this extra bookkeeping helps us determine which **retire** operation the handle belongs to. Now that each occurrence of h in the retired list has a different tag, when an **ejectAll**$_p$ operation moves a few copies of h from p's retired list to p's free list, we need to define which copies of h get moved as they are no longer identical. The **ejectAll**$_p$ operation begins by taking a snapshot of p's retired list and then scanning the announcement array. Suppose it sees s copies of h in p's retired list and t copies of h in the announcement array, then it moves exactly $s - t$ copies from p's retired list to p's free list. Among the s copies of h that the **ejectAll**$_p$ sees in p's retired list, at least $s - t$ of them are tagged with **retire** operations that the **ejectAll**$_p$ did not see in the announcement array. These are the handles that get moved to p's free list (if there are more than $s - t$ of such handles, an arbitrary subset of size $s - t$ is chosen). Now when an **eject**$_p(h)$ E removes a handle from p's free list, we define $g(E)$ to be the **retire**(h) operation tagged to that handle.

The function g that we constructed is injective because each tag gets added once to a retired list so after it is removed from the free list, it can never be removed again. Also, by the way **retire**s and **eject**s are linearized, we can see that $g(E)$ is always linearized before E. All that is left is to verify that whenever $f(A) = g(E)$, the **acquire** A is inactive at the linearization point of the **eject** E. Let R be the **retire** operation that both A and E are mapped to and let p be the process that performed E. Suppose for contradiction that A is still active at the linearization point of E. We can see from the linearization points of **acquire** and **release** that whenever A is active, the announcement array contains the handle announced by A. This handle is logically tagged with R. A is linearized before R by definition of f, so the handle tagged with R appears continuously in the announcement array between the linearization points of R and E. This would prevent any **ejectAll** operation between R and E from moving any handle tagged with R from p's retired list to its free list. Therefore E could not have removed a handle tagged with R from p's free list which contradicts the fact that g maps E to R.

The time for **acquire**, **release**, and **retire** are constant. The time for **eject** is constant in expectation because it may perform a constant number of operations on a process local hash table. If an **eject** is called after each **retire**, each process can have at most $O(K)$ in its retired list, $O(K)$ in a partially completed **ejectAll** and, $O(K)$ in the results from the previous **ejectAll** that have not been ejected yet. Therefore, our algorithm ensures that there are always no more than $O(KP)$ **retire**s that have not been ejected. Also, if **eject** is called after each **retire**, then the overall space usage is bounded by $O(KP)$ (recall that we assume $K \geq P$) because $O(K + P^2)$ space is used to implement the K `Destination` objects in the announcement array and $O(PK)$ space is used to store handles that have been retired but not ejected. The acquire-retire implementation only uses atomic single word read, write and CAS.

6.7 Evaluation

In this section, we provide an experimental evaluation of our C++ library across two benchmark setups. First, we compare its performance to other implementations of reference-counted pointers. Second, we compare our approach to the performance of manual SMR techniques. We performed some preliminary experiments using the wait-free **acquire** algorithm, and found that it was as fast as the lock-free one after applying a fast-path slow-path methodology [97], but since the performance was mostly determined by the fast path, we decided to use the simpler lock-free implementation for the rest of the experiments.

Setup. We ran our experiments on a 4-socket machine with 72 physical cores in total (Intel(R) Xeon(R) E7-8867 v4, 2.4GHz), 2-way hyperthreading, and 45MB L3 cache. The machine's interconnection layout is fully connected meaning that all four sockets are equidistant from each other. We interleaved memory across sockets using `numactl -i all`. For scalable memory allocation we used the jemalloc library [63]. All of our experiments were written in C++ and compiled with g++ version 9.2.1 on optimization level O3. Our experiments vary the number of threads from 1 to 200, which serves to also measure the effect of oversubscription since our hardware supports up to 144 with hyperthreading.

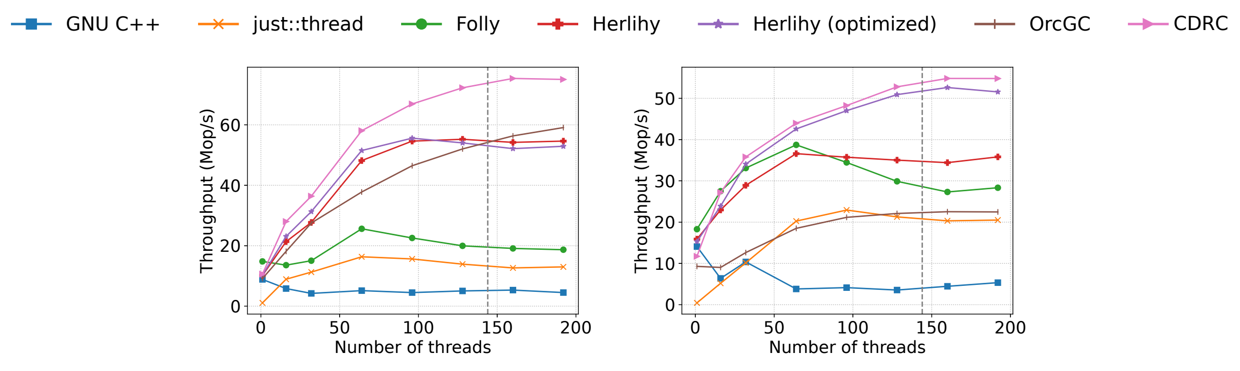
6.7.1 Comparison of Reference-Counting Techniques

We compare with implementations from four different libraries: the `atomic_` free functions for `shared_ptr`[2] from libstdc++ (The GNU C++ library [106]), Anthony William's just::thread library [169], Facebook's Folly library [64], and OrcGC [46]. The implementation in libstdc++ is lock-based whereas the others are lock-free. Both just::thread and Folly use something similar to the split reference count technique described in [168, Chapter 7.2.4]. We also implemented two reference-counted pointers based on Herlihy et al. [87]. The first follows their approach as closely as possible, while the second is an improved version that we optimized. Specifically, we replaced some of the CAS loops in the original algorithm with fetch-and-add and fetch-and-store instructions where applicable to improve performance.

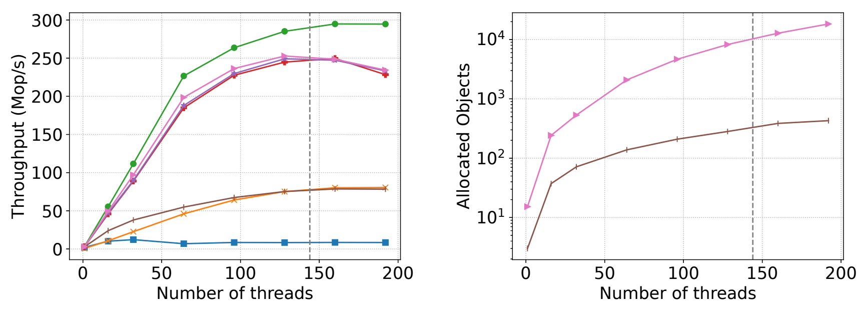
Microbenchmark #1: Load/Store Throughput. We maintain an array of N shared memory locations, each storing an atomic reference counted pointer to a 32-byte object. The array is padded so that each pointer is on a different cache line. Each thread picks a memory location uniformly at random and performs either a **load** or a **store**. Threads perform a **store** with probability p_s and a **load** with probability $1 - p_s$. Before a **store**, the thread allocates a new reference-counted pointer to a new object to be stored. After a thread performs a **load**, which increments the reference count, it reads the value being pointed to, and then destructs the loaded pointer. We show results for $N = 10$, a highly contended workload, and $N = 10M$, a workload with almost no contention. We run each experiment for 5 seconds, which was sufficient for reaching steady state performance, and report the total throughput of **load**s and **store**s averaged across 5 runs.

Results. The results of these experiments are depicted in Figures 6.8(a)–6.8(c). In the high-contention workloads (6.8(a)–6.8(b)) our implementation (CDRC) consistently outperforms the others, particularly on the load-heavy workload. Though Folly and just::thread use similar a similar technique, we found that Folly's implementation consistently outperforms just::thread. This is because Folly's implementation is highly optimized. For example, they pack a 48-bit

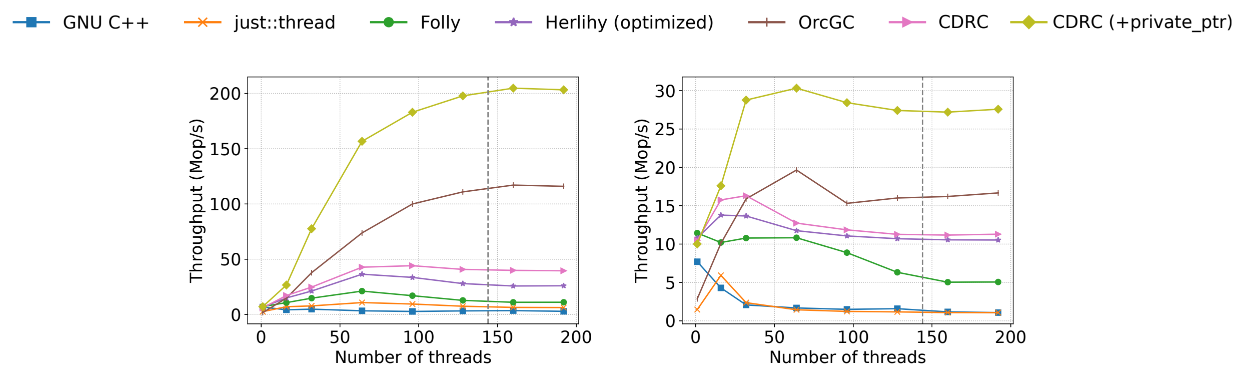
[2]At the time of writing, the latest C++ standard has deprecated these free functions and replaced them with specializations of `std::atomic`. However, neither libstdc++ or libc++ have yet provided an implementation.



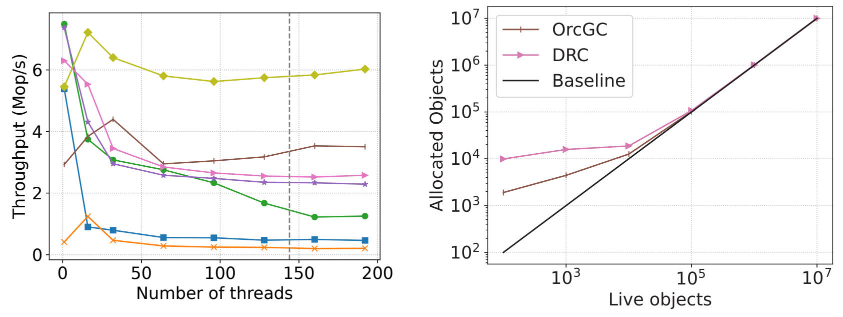
(a) $N = 10$, 10% stores (b) $N = 10$, 50% stores



(c) $N = 10^7$, 10% stores (d) Average allocated objects



(f) $N = 10$, 1% pushes/pops (g) $N = 10$, 10% pushes/pops



(h) $N = 10$, 50% pushes/pops (i) Average allocated objects

Figure 6.8: Benchmark results comparing reference-counted pointer implementations. Figures 6.8(a)–6.8(d) compare raw load/store throughput and memory usage. Figures 6.8(f)–6.8(i) compare throughput and memory when used to implement a concurrent stack.

pointer and a 16-bit counter into a single word to avoid the double-word-width CAS used by just::thread. The libstdc++ implementation achieves little if any observable speed up after 16 threads because it uses a set of 16 global locks. The second-best overall competitor is Herlihy's algorithm, our improved version of which comes close to the performance of our algorithm on the store-heavy workload. Although it does not exhibit the strongest throughput, OrcGC shows consistent scaling. On the read-heavy workload, it catches up to the performance of Herlihy at 144 threads, and takes over second place once oversubscription is entered. On the store-heavy workload, however, OrcGC is consistently outperformed by both Folly and Herlihy.

On the low contention workload, Folly is the winner, while Herlihy and our algorithm come in second. just::thread and OrcGC trail behind, and libstdc++ exhibits no scaling at all. Folly's performance is attributable to the fact that, under low contention, the work performed by the deferred algorithms to acquire and protect the pointer is almost always unnecessary. OrcGC's performance on the store-heavy and low-contention workloads compared to its stronger earlier performance on the load-heavy workload suggest that its store operation is particularly expensive. This can be explained by the fact that its retire operation, which will be invoked on each store, performs $O(P)$ work, while ours and Herlihy perform constant expected work.

The tradeoff is that our approach and that of Herlihy use more memory. They may defer up to $O(P^2)$ reclamations, while OrcGC defers at most $O(P)$ reclamations, and the other schemes perform no deferred reclamation and always reclaim immediately. In Figure 6.8(d), we show the average memory usage in terms of the number of allocated objects against the number of threads. The average number of objects allocated for our algorithm is approximately $0.5P^2$, while the number allocated by OrcGC is approximately $3P$, which matches the theoretically expected bounds.

Microbenchmark #2: Concurrent Stack. We implemented a concurrent stack using the code shown in Figure 6.1, but also supporting a *find* operation that takes as input, a value, and searches the stack, returning true if that value is present. Implementations that do not support **load_private** perform a **load** instead. We maintain an array of $N = 10$ concurrent stacks, each padded to its own cache line. Every stack initially has 20 elements. Threads perform a find on a uniformly random stack with probability p_f, or, with probability $1 - p_f$, a pop from a uniformly random stack followed by a push of the popped value onto another uniformly random stack (possibly the same one). If the popped stack was empty, nothing is pushed. We show results for $p_f = 0.01, 0.1, 0.5$, indicating read-heavy, read-mostly, and update-mostly workloads.

Results. The results are depicted in Figures 6.8(f)–6.8(h). We test our CDRC algorithm both with and without private pointers. The clearest takeaway from these experiments is that private pointers provide tremendous benefits, particularly on read-heavy workloads. Recall that OrcGC also employs a technique similar to private pointers, which is why it, too, outperforms the other methods. CDRC without private pointers outperforms the remaining implementations, but by a smaller margin. At 128 threads, private pointers improve the throughput of the read-heavy workload by 1.7x compared to OrcGC, 5x compared to CDRC without private pointers, 7x compared to our optimized implementation of Herlihy's algorithm, and 16x compared to Folly. On the update-mostly workload, our algorithm still outperforms the other implementations by at least 2x, due to finds not creating contention with updates.

Lastly, in Figure 6.8(i), we show the memory usage in terms of the number of allocated nodes with respect to the number of live nodes (the total number of nodes in all of the stacks). The number of threads in this experiment was fixed at 128. As the number of live nodes increases, the number of allocated nodes is asymptotic to the number of live nodes, indicating that the memory overhead of the schemes is indeed additive, and not proportional to the number of live nodes.

6.7.2 Comparison to Manual SMR Techniques

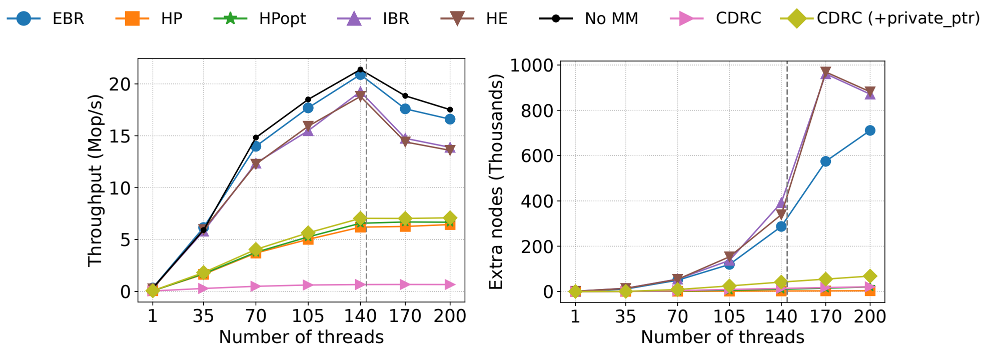
We compare our reference-counting technique with four different manual SMR techniques, hazard pointers (HP) [117], hazard eras (HE) [142], two-global-epoch IBR [167] and epoch-based reclamaion (EBR) [71] applied to three different lock-free data structures: Harris-Michael list [80, 116], Michael hash table [116], and Natarajan-Mittal tree [122]. When applying our technique, we use private pointers for the short-lived references that processes hold onto while traversing the data structure. In the Natarajan-Mittal tree, each process holds onto at most five private pointers at a time, and in the list and hash table, each process holds onto at most three. To measure the benefits of private pointers, we also benchmark our implementation without them, using only `shared_ptr`, which increments reference counts eagerly. As a baseline, we also measure the performance of each data structure when no memory reclamation is performed, meaning that nodes are never freed at all.

Benchmarks. We leveraged the IBR benchmark suite [167] which contains implementations of HP, HE, IBR and EBR applied to the three data structures. In Section 6.8, we identify some bugs in the IBR benchmarking suite related to incorrectly applying these memory reclamation techniques. For our benchmarks, we fixed all of them except the last one, which only applies to the Natajaran-Mittal tree when used with HP, HE, or IBR. Fixing this would required significant modifications to the data structure, and would only slow down the performance of these SMR techniques due to the extra restarts. Therefore, these experiments depict a generous estimate of how HP, HE, and IBR would perform when correctly applied to the Natarajan-Mittal tree. We also optimized the throughput of the HP implementation by reducing the number of times the announcement array is scanned. While this significantly improves throughput in some cases, it does so at the cost of a slight increase in memory.

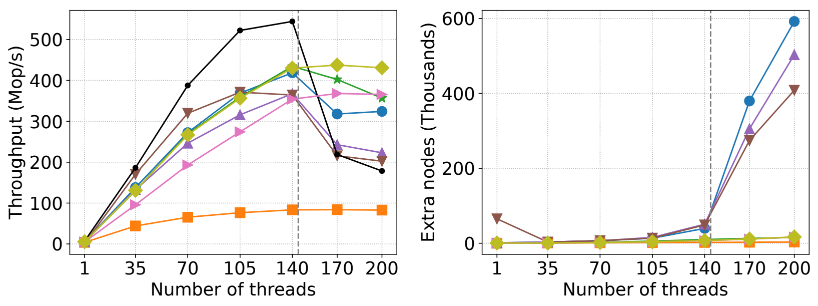
For each data structure, we tried various sizes and update frequencies. For example in Figure 6.9(c), we initialized the BST with 100K keys and each process performed 10% update operations (half insert, and half delete) using a key chosen uniformly randomly from the range $[0, 200K)$. The remaining 90% of operations were lookups. For the hash table experiments, we initialized the number of buckets so that the average load factor is one.

Results. The results of these experiments are shown in Figures 6.9 and 6.10. In each pair of graphs, throughput is plotted on the left and space overhead is plotted on the right. Space overhead is measured by calculating the number of nodes that were removed from the data structure and not yet freed.

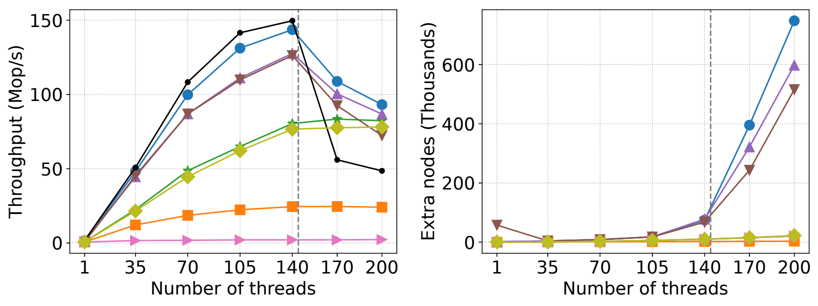
We found that using private pointers is crucial for getting reference counting to scale on many of these lock-free data structures. It improves performance by up to 40× in Figure 6.10(b) and a minimum of 1.2× in Figure 6.9(b). This optimization is what allows automatic reference



(a) List. N=1000, updates=10%. Throughput (L), Memory (R)

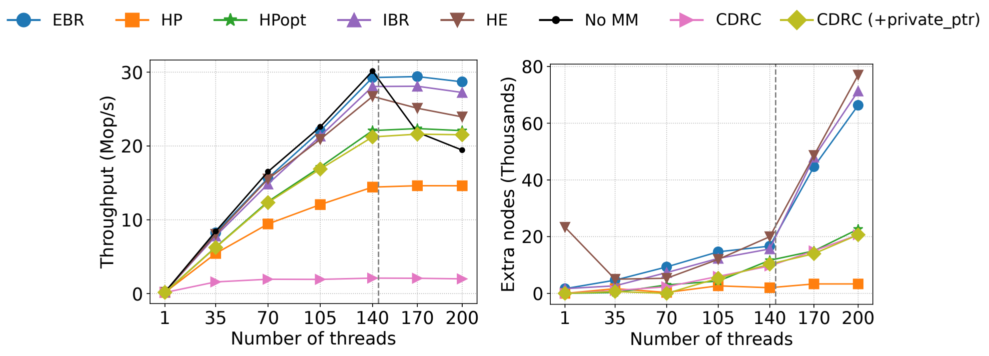


(b) Hash table. N=100K, updates=10%. Throughput (L), Memory (R)

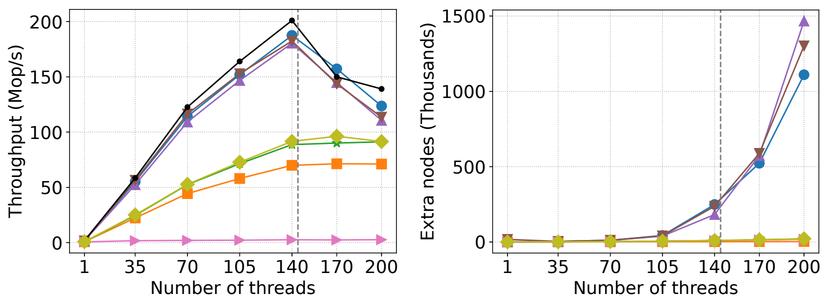


(c) BST. N=100K, updates=10%. Throughput (L), Memory (R)

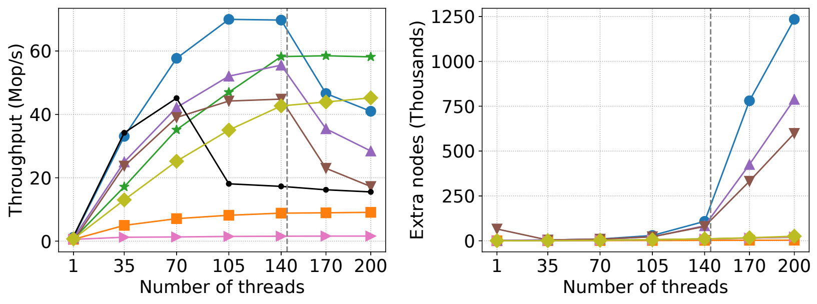
Figure 6.9: Benchmark results comparing deferred reference counting with manual SMR techniques. Figure 6.9(a) shows results for a Harris-Michael list, Figure 6.9(b) for a Michael hash table, and Figure 6.9(c) for a Natarajan-Mittal tree.



(a) BST. N=100M, updates=10%. Throughput (L), Memory (R)



(b) BST. N=100K, updates=1%. Throughput (L), Memory (R)



(c) BST. N=100K, updates=50%. Throughput (L), Memory (R)

Figure 6.10: Benchmark results comparing deferred reference counting with manual SMR techniques for various workloads on a Natarajan-Mittal tree.

counting to be competitive with manual SMR. Overall, our reference-counting technique tends to closely match the throughput and space usage of optimized hazard pointers (HPopt). One exception is in the update-heavy workload of Figure 6.10(c) where the cost of reference-count increments and decrements during updates causes a 38% performance overhead.

We found that our technique generally performs very well on hash table workloads (one of which is shown in Figure 6.9(b)) because on average, each lookup acquires one private pointer, which is about as cheap as acquiring a HP or announcing an epoch during EBR. In this workload, for thread counts of 140 or higher, our technique actually outperforms all of the manual SMR techniques.

In general, our technique does not seem to be slowed down by over-subscription whereas HE, IBR, and EBR are often severely impacted. The memory usage of HE, IBR and EBR spike upwards during over-subscription because one stalled thread can prevent a lot of nodes from being collected.

In most cases, our throughput is 1.2-2.5× slower than EBR, but we experience 3-61× less memory overhead. The only exception is the linked list workload in Figure 6.9(a), where we are up to 5.1× slower than EBR, but in exchange, we waste 210× less memory on 200 threads, and 6.5× less on 140 threads. Our memory usage is always within a factor 3 of HPopt, which indicates that having P^2 delayed decrements usually translates to holding onto about P^2 extra nodes for these data structures.

These results show that automatic reference counting, when implemented efficiently, can perform competitively with manual memory reclamation techniques. Furthermore, whenever manual techniques outperform our algorithm, our algorithm uses significantly less space.

6.8 Usability Difficulties of Manual SMR

Applying manual memory reclamation techniques to concurrent data structures can be non-trivial and difficult to get right, even for expert users, often leading to bugs that are not caught for a long time. In this section, we will discuss some recurring bugs that we have discovered in research code while working on memory reclamation. We emphasize that these bugs exist in the applications of these memory reclamation techniques, not in the techniques themselves.

Correctly Calling Retire. While some manual techniques are more difficult to apply than others, one thing that they have in common is the need for the user to determine when an object is no longer reachable from the shared data structure and explicitly call `retire` on this object. This can be challenging in a concurrent setting. For example, if there are two pointers in shared memory to an object and the pointers are concurrently cleared by two different processes, it is not clear which process should be the one to call `retire` or how the process even learns that the other pointer has been cleared. Another issue is that it is easy to forget to retire a node, especially when there are concurrent operations involved. For example, in the Natarajan and Mittal tree [122], the delete operation marks an internal node for deletion, and then calls a cleanup procedure which performs a CAS removing the node. A common mistake is to only retire a single internal node after this CAS. However, in the presence of concurrent deletes, this CAS can potentially remove a long chain of marked nodes, all of which need to be retired. This exact memory leak can be found in the artifacts of several papers [46, 48, 73, 125, 167], some of which are specifically about safe memory reclamation.

Restarts. An important detail that is sometimes missed is that many of these manual reclamation techniques (HP, HE, WHE, IBR) often require significant changes to the original concurrent data structure in order to be applicable. To protect an object using one of these techniques, a process has to first announce either a pointer to the object or an epoch, and then verify that the object has not been retired. If there is no way to verify this, then the object could have already been freed before the announcement happened, so it is not safe to access. In this case, some sort of fall back plan is needed and this usually involves aborting and restarting the operation. The IBR and WHE benchmark suites applied HP, HE, WHE, and IBR to the original Natarajan and Mittal tree without additional restarting, which leads to unsafe memory accesses.

We conclude this section by reiterating our premise that manual SMR techniques are easier to misuse than automatic ones, so most users should prefer to rely on automatic memory reclamation.

6.9 Discussion and Conclusion

In this work, we designed, analyzed, and evaluated a new technique for automatic memory reclamation for non-garbage-collected languages based on a novel combination of SMR techniques and reference counting. We showed that our technique is theoretically more efficient than existing methods, and demonstrated that it is also practical by implementing it as a library for C++ and comparing it to a range of existing schemes, both automatic and manual. Our method performs strongly against existing automatic techniques, improving performance by up to a factor of 16 when compared against state-of-the-art open source and commercial reference-counted pointers. Against manual SMR techniques, it remains competitive, achieving similar throughput and memory consumption to hazard pointers, and usually performing within a factor of 1.2-2.5× against the fastest manual techniques that consume unbounded amounts of memory.

A limitation of our automatic memory reclamation technique that we inherit from reference counting is that an object cannot be collected while it is part of a reference cycle. There are many approaches to deal with cycles (e.g. weak pointers) and it would be interesting to explore incorporating those into our technique.

Lastly, although we have applied the acquire-retire framework specifically to reference counting, we believe that the framework on its own is also important. By considering resources in general, and supporting multiple retires on the same resource, our interface generalizes previous ones, which focused mostly on memory-reclamation. We believe it will find a range of applications beyond reference counting and possibly even beyond memory reclamation.

Chapter 7

Turning Manual Memory Reclamation into Automatic Reference Counting

7.1 Introduction

The previous chapter presents CDRC, a concurrent reference counting algorithm that is significantly faster than previous automatic SMR schemes and achieves close to the performance of Hazard Pointers (HP) in practice. However, it still has up to a factor of two performance degradation relative to manual memory reclamation via EBR. The main issue is the use of protected-pointer techniques which require extra memory barriers on every read (even if the count is not incremented).

In this chapter, we show that reference counting can be nearly as fast as *any* manual technique while using a similar amount of memory (in most cases), thus showing that the ease-of-use of automatic approaches comes at no significant cost to practical performance. This approach is based on CDRC, which combines reference counting and hazard pointers in a novel way. Unlike traditional methods which use hazard pointers to protect a block of memory from being freed, the key insight in CDRC is that hazard pointers can be used to protect *the reference count itself* from being decremented. This simple insight leads to two crucial patterns. First, *deferred decrements* allow increments to proceed without fear of racing with a decrement that might set the counter to zero, thus solving the read-reclaim race. Second, and critically for performance, being able to temporarily protect the reference count from decrements enables readers to safely read the managed object without fear of its destruction and without the performance cost of incrementing the reference count.

One of the contributions of this chapter is generalizing the CDRC technique so that the hazard pointer scheme can be replaced with just about any standard SMR scheme to yield an automatic version of that scheme with a similar performance profile. We apply this to three (very different) state-of-the-art manual techniques, EBR, IBR and Hyaline, to yield automatic versions of all three. To the best of our knowledge, this is the first time reference counting has been combined with any manual technique outside of variations of hazard-pointers. The resulting algorithms are all lock-free, assuming that the SMR scheme being automated is lock-free.

As a second contribution, we show how this framework can be extended even further to support lock-free atomic weak pointers that also allow safe reads without incrementing the reference count. We use them to implement a concurrent doubly-linked-list based queue [141], and show that our implementation is several times faster than the only other lock-free atomic weak pointer that we are aware of [169].

A key challenge with weak pointers is supporting the upgrade to strong pointers efficiently. This requires being able to atomically increment the reference count only if it is not already zero. This operation is typically implemented using a CAS-loop [106] which takes up to $O(P)$ amortized time per process if P processes perform this upgrade at the same time. Instead, we show how to implement a so-called sticky counter primitive that supports an *increment-if-not-zero* operation so that reading and incrementing/decrementing take only $O(1)$ time in the worst case.

Contributions.

- We show that a wide range of manual SMR techniques can be made automatic using reference counting.
- We show experimentally that our automatic techniques have similar throughput and memory usage to their manual counterparts. (This represents a 2x-3x throughput improvement over existing concurrent reference counting implementations.)
- We show how to extend our reference counting techniques to efficiently support atomic weak pointers.
- To do so, we implement a theoretically and practically efficient sticky counter primitive.
- We show that our weak pointers significantly outperform existing weak pointers in practice.

Outline. In Section 5.2, we introduce some important background information and we defer a broader discussion of related work to the end of this part. Section 7.3 describes a general technique for making manual memory reclamation automatic. In Section 7.4, we show how to extend our algorithms with support for weakly reference-counted pointers to handle reference cycles. An experimental evaluation of the techniques described in this chapter is presented in Section 7.5. Finally, we conclude in Section 7.6.

7.2 Background

Manual SMR. Most manual SMR schemes have similar interfaces built around a common set of operations. These operations include:

- **retire**(x): Indicate that an allocated object x is no longer reachable by the program, i.e., that it is safe to delete after all readers currently reading it are finished.
- **eject**(): Returns a previously retired object that is now safe to delete. The caller should then free this object.

The retire operation is the critical one; it is what replaces completely manual memory management (explicit freeing). A retire operation is essentially a “delayed free”. Rather than being freed immediately, the object is freed once any lingering readers have finished with it. The eject operation is optional and is often performed implicitly by retire, but separating the two can allow the programmer greater control over exactly when or how memory is freed.

The difference between *protected-pointer* and *protected-region* techniques is in how they determine when the lingering readers have finished with a retired object, making it safe to free. Protected-region techniques implement the following pair of operations:

- **begin_critical_section**(): Indicate the beginning of a read critical section.
- **end_critical_section**(): Indicate the end of the current read critical section.

For correctness, all reads of objects that are protected by the SMR scheme must be performed while inside a read critical section. A retire operation is able to deduce that a retired object x is safe to eject once all critical sections that were active at the time of its retirement have ended. Protected-pointer techniques use the following operations instead:

- **acquire**(m): Indicate the intention to read the contents of a shared pointer located at the memory location m, and return the current value of the shared pointer.
- **release**(p): Indicate that the pointer obtained from a shared location by **acquire** is no longer being read.

All reads of objects that are protected by the SMR scheme must be done so via an acquire operation, and ended by a corresponding release operation. A retire operation is then able to safely deduce that a retired object x is safe to eject once all active acquires of it at the time of its retirement have been released. Note that in many protected pointer schemes such as hazard-pointer and pass-the-buck, the acquire operation can fail, forcing the program to retry or take a data structure specific fallback plan.

The difference between protected-pointer and protected-region techniques is that protected-region techniques prevent *all* objects from being ejected during their read critical sections, while protected-pointer techniques are more granular and only protect the objects actually being read. Protected-region techniques are therefore usually faster since they require less bookkeeping, but accumulate more garbage because they overprotect objects from being ejected.

7.3 Making Manual SMR Automatic

In this section, we describe how to make manual SMR automatic by combining it with reference counting. This section extends the CDRC approach from Chapter 6 which uses a hazard-pointer-like technique called acquire-retire to delay reference count decrements until they no longer race with increments. The insight in this section is that this approach would work for virtually any manual SMR technique, not just hazard-pointers. Note that the process of converting from manual to automatic SMR is not automatic, but we present an easy-to-apply framework and show examples of how to use it.

To generalize CDRC, we first generalize its acquire-retire interface and then show that this generalized interface can be implemented from a wide range of manual techniques. Then we show how to implement concurrent reference counting using this generalized interface.

```
class AcquireRetire<T> {
// Allocate object of type T
Function alloc(): T*

// Delays destructing ptr
Function retire(T* ptr): void

// Returns a previously retired pointer
// that is no longer protected.
Function eject(): optional<T*>

Function begin_critical_section(): void
Function end_critical_section(): void

// Reads a pointer from shared memory and protects it.
// Can only protect one pointer at a time.
Function acquire(T** ptraddr): pair<T*, Guard>

// Reads a pointer from shared memory and tries to protect it
// Can fail and return ⊥.
Function try_acquire(T** ptraddr): optional<pair<T*, Guard>>

// Releases protection
Function release(Guard guard): T* };
```

Figure 7.1: Generalized acquire-retire interface.

7.3.1 Generalized Acquire-Retire Interface

The generalized acquire-retire interface shown in Figure 7.1 has several advantages over the original. The original interface is well-suited for capturing protected-pointer SMR techniques (because `acquire` protects a specific pointer), but not for capturing other types of SMR techniques. We added three new methods to make the interface more general: `alloc`, and `begin_` and `end_critical_section`. Adding `alloc` to the interface is important for techniques like IBR and HE, which tag each object with a birth timestamp on allocation.

Beyond generality, another benefit of the interface in Figure 7.1 is that it allows for a clean implementation of private pointers. In CDRC, supporting private pointers requires reaching into the internals of the acquire-retire implementation. So unlike the rest of the reference counting algorithm from Chapter 6, the part that implements private pointers only works for the specific implementation of acquire-retire presented in that chapter. We fix this problem by breaking their `acquire` into two operations, an `acquire` and a `try_acquire`. Both operations return a pointer as well as a guard variable that protects the pointer. The pointer can be unprotected at any point by passing the guard variable to `release`. In HP and HE, this guard variable would be a pointer to the announcement slot that protects the pointer. `acquire` can only protect one pointer at a time, so the user must alternate between calling `acquire` and `release`. `try_acquire` on the other hand can protect multiple pointers with different guards. However `try_acquire` may fail and return ⊥ if it runs out of guards (e.g. running out of hazard-pointers). We use `try_acquire` to implement `private_ptrs` in a black box manner in Section 7.3.4.

```
class AcquireRetireEBR<T> {
  using Guard = void; // empty type, never used
  using Epoch = int;
  Epoch ann[P]; // initialized to INT_MAX
  Epoch curEpoch = 0;
  thread_local List<pair<T*, Epoch>> retired;

  T* alloc() { return new T(); }
  void begin_critical_section() { ann[pid] = curEpoch; }
  void end_critical_section() { ann[pid] = INT_MAX; }
  void release(Guard guard) {}

  pair<T*, Guard> acquire(T** ptraddr) {
    return [*ptraddr, void]; }

  optional<pair<T*, Guard>> try_acquire(T** ptraddr) {
    return [*ptraddr, void>]; }

  // retire + eject implemented as in Figure 2 of [167] };
```

Figure 7.2: Generalized acquire-retire implemented with epoch-based-reclamation. We assume each process knows its process id *pid*.

Lastly, just like in the original acquire-retire interface, the `retire` operation in Figure 7.1 takes as input a pointer which will be returned by a future `eject` operation when it is no longer protected.

7.3.2 Implementing Generalized Acquire-Retire

This new acquire-retire interface can be implemented from almost any manual SMR technique. Figures 7.2 and 7.3 show implementations from EBR and IBR, respectively. In this section, we will discuss some general patterns in these implementations. Most manual SMR algorithms combine the functionality of `retire` and `eject` into a single `retire` operation, but this is easy to split into two operations. A more important difference is that manual SMR is typically used to delay freeing objects. So instead of returning retired pointers to the user, their `retire` function calls `free` on pointers that are no longer protected. We require pointers to be returned to the user because our `retire` can be used to delay arbitrary operations on the pointer, for example decrementing the pointer's reference count. In our implementation of weak pointers in Section 7.4, we use three instances of `AcquireRetire`, each delaying a different type of operation.

Another reason for having `eject` return a pointer instead of directly applying the delayed operation is to prevent `eject` from recursively calling itself. For example, if the delayed operation is a reference count decrement, then this might trigger recursive reference count decrements, which might lead to recursive calls to `eject`. The `eject` operation is not guaranteed to behave correctly if called recursively, so we disallow this possibility by not applying the delayed operation inside the `eject`. The final difference between our `retire` and the one supported by existing SMR techniques is that we allow a pointer to be retired any number of times before it is ejected a single time. Luckily, most SMR algorithms work properly in this kind of situation even

```
class AcquireRetireIBR<T> {
  using Guard = void; // empty type, never used
  using Epoch = int;
  Epoch emptyann = INT_MAX;
  Epoch beginAnn[P], endAnn[P]; // initialized to emptyann
  Epoch curEpoch = 0;
  thread_local Epoch prev_epoch = emptyann;
  thread_local int counter = 0;

  void begin_critical_section() {
    beginAnn[pid] = endAnn[pid] = prev_epoch = curEpoch; }
  void end_critical_section() {
    beginAnn[pid] = endAnn[pid] = emptyAnn; }
  void release(Guard guard) {}
  class Tagged<T> { Epoch birthEpoch; T t; };

  T* alloc() {
    Tagged<T>* taggedObj = new Tagged<T>();
    taggedObj->birthEpoch = curEpoch;
    if(counter++ % epoch_freq == 0) curEpoch.fetch_add(1);
    return addressof(taggedObj->t); }

  pair<T*, Guard> acquire(T** ptraddr) {
    while(true) {
      T* ptr = *ptraddr;
      Epoch cur_epoch = curEpoch;
      if(prev_epoch == cur_epoch) return [ptr, void];
      else endAnn[pid] = prev_epoch = cur_epoch; } }

  optional<pair<T*, Guard>> try_acquire(T** ptraddr) {
    return acquire(ptraddr); }

  // retire + eject implemented as in [167] };
```
Figure 7.3: Generalized acquire-retire implemented with interval-based-reclamation (specifically, 2GEIBR).

though they were not designed with it in mind. Protected-pointer approaches sometimes need to be modified to keep track of the number of times a pointer is `retired` and `acquired`. `eject` also has to be modified so that it returns only the pointers that have been `retired` more times than `acquired`. No such modifications are needed for protected region approaches.

Next, we focus on how to implement `acquire`, `release`, and `try_acquire`. For protected-region SMR techniques like EBR, and Hyaline, these operations are trivial to implement because the critical section on its own is enough to protect all the pointers returned by `acquire`. So `acquire` and `try_acquire` simply load the pointer and `release` is a no-op. For protected-pointer approaches like HP and PTB, `try_acquire` has to look for an empty announcement slot to act as the guard. If all announcement slots are in use, then `try_acquire` fails, returning $\perp$. For `acquire`, we reserve a special guard / announcement slot that cannot be used by `try_acquire`. This ensures that `acquire` always succeeds but it means that only one pointer can be protected by `acquire` at a time.

Finally, the operations for beginning and ending a critical section are implemented in the exact same way as in the corresponding manual SMR technique. So for EBR, they would just announce and unannounce an epoch, and for protected-pointer approaches, they would be no-ops.

7.3.3 Defining Correctness

Just like with the original acquire-retire interface, the tricky part of defining correctness for the generalized version is handling the case where a pointer gets retired multiple times before any copy gets ejected. Fortunately, we can use the original correctness definition with just some small modifications. The idea behind the original definition is to map acquires to retires and ejects to retires such that if an acquire and an eject get mapped to the same retire, then the acquire must be inactive by the time the eject is executed. This formalizes the intuition that a pointer can only be returned by eject if it is not protected by any active acquire. We begin by defining what it means for an acquire to be *active.*

Definition 5 (active vs. inactive acquires)**.** *We say that an* `acquire` *or a successful* `try_acquire` *is* active *between when it was invoked and when the guard it returns is passed to* `release`*. After its guard is released, we say it is* inactive*.*

Our acquire-retire interface imposes some restrictions on how it can be used. These restrictions are captured in the following definition of *proper executions.*

Definition 6 (proper execution)**.** *We say that a concurrent execution involving acquire-retire operations is* proper *if (1) each active acquire is contained in a critical section, (2) each guard returned by* `acquire` *or* `try_acquire` *is passed to* `release` *at most once, and (3) a process cannot call* `acquire` *while its previous* `acquire` *is still active.*

The first property in Definition 6 is easy to ensure by beginning a critical section before any calls to acquire and making sure all acquires are inactive before ending the critical section. The third property just says that acquire can only be used to protect a single pointer at a time. Now we are ready to formally define the sequential specifications of acquire-retire.

```
class private_ptr<T> { T* ptr; optional<Guard> guard; };

AcquireRetire<T> ar;

private_ptr<T> atomic_shared_ptr<T>::load_private() {
  auto ptr, guard = ar.try_acquire(addressof(this->ptr));
  if(guard != ⊥) return private_ptr<T>(ptr, guard);
  ptr, guard = ar.acquire(addressof(this->ptr));
  increment(ptr); // increment reference count
  ar.release(guard);
  return private_ptr<T>(ptr, ⊥); }

void private_ptr<T>::release() {
  if(this->guard != ⊥) ar.release(this->guard);
  else decrement(this->ptr); }

void begin_critical_section() { ar.begin_critical_section(); }
void end_critical_section() { ar.end_critical_section(); }
```
Figure 7.4: Implementing private pointers using the generalized acquire-retire interface from Figure 7.1.

Definition 7 (acquire-retire). *Any proper, concurrent execution can be linearized to a sequential history with the following guarantees:*

- *Successful* `try_acquire(pptr)` *and* `acquire(pptr)` *operations return the pointer currently stored in* $*pptr$.
- *Let f be a function that maps each* `acquire` *returning p and each successful* `try_acquire` *returning p to either a later* `retire(p)` *or* $\perp$. *Let g be an injective (one-to-one) function that maps each* `eject` *returning p to an earlier* `retire(p)`. *For all f, there is a g such that whenever $f(A) = g(E)$, the* `acquire` *or* `try_acquire` *A is inactive by the time* `eject` *E is executed.*

7.3.4 Concurrent Reference Counting

Using the generalized acquire-retire interface, we can implement concurrent reference counting in much the same way as CDRC. The main difference is in our implementation of `private_ptr`s shown in Figure 7.4. The code for the other two reference-counted pointer types, `atomic_shared_ptr` and `shared_ptr`, remains the same except for some minor updates to use the new acquire-retire interface.

We support private pointers by implementing an operation called `load_private` which loads an atomic shared pointer and creates a `private_ptr`, and by implementing a `release` operation which destructs a `private_ptr`. `load_private` first tries to take the fast path which consists of protecting the pointer with just a `try_acquire`. If this `try_acquire` fails, then it reverts to the slow path which consists of protecting the pointer using an `acquire`, then incrementing the reference count of the pointer, and then releasing the previous acquire since the pointer is now protected by the incremented reference count. In the slow path, `load_private` then constructs and returns a `private_ptr` with its `guard` field set to $\perp$ to indicate that the slow path was taken. A `private_ptr`'s destructor calls `ar.release()` if it was constructed via the fast path and `decrement` otherwise. As long as a process does not hold on to too many `private_ptr`s, `load_private` will always take the fast path and not perform any reference count updates. This is why `private_ptr` can be cheaper than `shared_ptr`s.

This is different from CDRC's `load_private` implementation which only works for a specific acquire-retire implementation based on hazard-pointers. In CDRC, `load_private` first looks for an empty announcement location and if all of them are taken, it evicts one of the announcement hazard pointers and increments the reference count of the evicted pointer to ensure that it stays protected. Then `load_private` uses the newly emptied announcement location to protect the pointer it reads.

Another difference from CDRC's implementation is that we require all racy[1] reads and writes on atomic shared pointers as well as all private pointer lifetimes to be contained in a critical section. When applying our reference counting algorithm to a concurrent data structure, this requirement can be satisfied by wrapping each data structure operation in a critical section and only holding on to private pointers during the operation.

7.4 Weak Pointers

The second classical drawback of reference counting is its inability to clean up garbage that contains cyclic references. A common approach to mitigate this issue at the library level is to include a "weak pointer" type. Weak pointers complement shared pointers (or "strong pointers") by holding a reference to a shared object without contributing to the reference count. If the reference count of the managed object reaches zero, it is destroyed, despite any weak pointers that may have a reference to it.

The advantage of weak pointers over raw pointers is that, unlike raw pointers, which are unsafe to follow if they might point to an already freed object, weak pointers can tell whether they point to a managed object that has already been destroyed. This is usually achieved by storing a second reference count that counts the number of weak pointers to the managed object. When the (strong) reference count reaches zero, the managed object is destroyed, but the control data containing the reference counts is kept intact until both the strong and weak reference counts reach zero. This allows weak pointers to safely check that the managed object is alive by checking that the strong reference count is non-zero.

The C++ standard library includes support for weak pointers, and, as of C++20, support for atomic weak pointers. However, currently the only standard library implementation of atomic weak pointers is Microsoft's STL [121], and it is lock-based. We know of one commercial implementation in the just::thread library [169]. We describe how our approach can be extended to efficiently support weak pointers.

7.4.1 Library Interface

We add the following types to the reference-counted pointer library. Figure 7.5 depicts the relationship between them.

- **atomic_weak_ptr**: Analogous to `atomic_shared_ptr`, an `atomic_weak_ptr` facilitates atomically loading, storing, and CASing a `weak_ptr` into a shared mutable location. In addition to

[1]Two operations are said to *race* if they both access the same atomic shared pointer and one of them is a write.

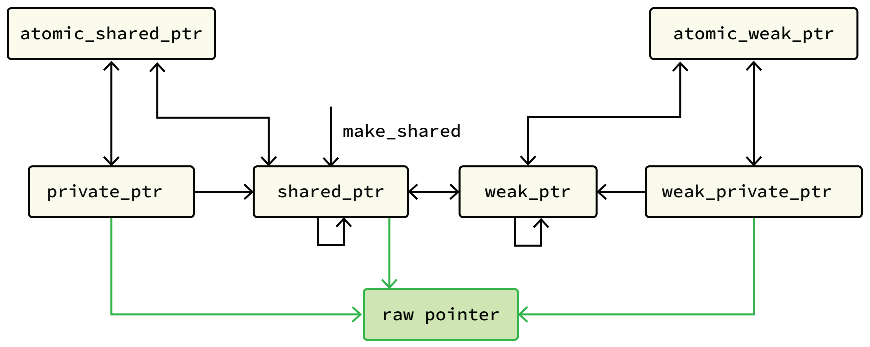


Figure 7.5: The managed pointer types in our library. Arrows between types denote that it is possible to store/load one type in/from the other, or that it is possible to convert from one type to the other. The three types `private_ptr`, `shared_ptr`, and `weak_private_ptr` can be safely followed/converted into raw pointers.

`load`, it also supports a `load_private` method, which grants safe local access to the managed object without modifying the reference count.

- **weak_ptr**: A `weak_ptr` is modeled after C++'s standard weak pointer. Unlike `shared_ptr`, a `weak_ptr` cannot be directly followed. To access the managed object, the `weak_ptr` must be upgraded to a `shared_ptr`. If the managed object has *expired*, the obtained `shared_ptr` will be null to indicate this.
- **weak_private_ptr**: A `weak_private_ptr` allows safe access to the object managed by the `atomic_weak_ptr` as of the time it was created, even if the reference count of the managed object reaches zero during its lifetime. Creating and reading a `weak_private_ptr` does not incur a modification to the reference count. A `weak_private_ptr` will be null if the managed object has expired at the time of its creation.

The subtle difference between a `weak_private_ptr` and a `private_ptr` is that a `private_ptr` guarantees that the managed object doesn't expire (has reference count at least one) throughout its lifetime, while a `weak_private_ptr` only guarantees that the managed object is safely readable, though it may expire (reach reference count zero) during the lifetime of the private pointer.

We first describe the main primitives needed to implement deferred reference counting with weak pointers. We then describe how to support the main operations on the various weak pointer types in our library.

7.4.2 Managing the Managed Object

First, to implement weak pointers, each managed object is augmented with a second reference count. We distinguish between the original (strong) reference count and the new (weak) reference count. When the strong reference count reaches zero, the managed object is ready to be destroyed. However, the control data attached to the managed object (the reference counts plus any extra scheme-specific metadata) cannot be destroyed and freed yet, because there might still exist weak pointers that attempt to access those fields. Only once both the strong and weak reference counters hit zero can the entire control block (the managed object plus the control data) be freed.

To correctly detect when both counters hit zero in the presence of concurrent updates, we use the standard trick [106, 121] of storing

$$\text{weak\_cnt} = \text{\#weak refs} + \begin{cases} 1 & \text{if \#strong refs } > 0 \\ 0 & \text{otherwise.} \end{cases}$$

When the strong count hits zero, it can destroy the managed object and decrement one from the weak count. To be precise, this destruction and corresponding decrement must be delayed in the presence of weak pointers. We will discuss this in Section 7.4.4. When the weak count hits zero, the entire control block is ready to be freed immediately.

In the strong-only setting, the reference count will only ever be incremented when there already exists at least one reference, and hence the increment can always be performed with a fetch-and-add operation. In the weak setting, however, it is possible that a weak pointer points to a managed object whose strong reference count could be decremented to zero at any moment. Attempting to increment the strong reference count with a fetch-and-add could therefore result in incrementing the counter from zero, thus resurrecting a dead object. Our algorithms therefore require an *increment-if-not-zero* operation, which can return false if the reference count is zero, and hence should not be incremented.

The increment-if-not-zero operation is traditionally implemented as a simple CAS loop, which continuously attempts to add one to reference count as long as it is not zero, or returns false otherwise. This results in the increment having lock-free but not wait-free progress. In the next section, we describe a simple, but to the best of our knowledge, novel implementation of a constant-time wait-free counter that supports the increment-if-not-zero operation. This data structure in general is sometimes referred to as a sticky counter. Specifically, our data structure implements an atomic counter that supports increment-if-not-zero, decrement, and load, all in constant time using single-word atomic instructions.

7.4.3 Wait-Free Increment-if-Not-Zero

Our algorithm can implement a b-bit wait-free counter using $b + 2$ bits, that is, we use two bits for bookkeeping purposes. The main idea is simple, we use the highest bit of the reference counter to indicate whether the reference count is zero. Any bit pattern in which the highest bit is set is interpreted as zero, and otherwise is not. Note importantly, that this means that the stored value being zero is not interpreted as the reference count being zero! The implementation is described below and depicted in Figure 7.6. This technique of using the high bits to store a flag above a counter is similar to that of Correia and Ramalhete [45] who implement reader-writer locks that store a count of the number of shared readers. Our technique generalizes theirs by allowing constant-time linearizable reads of the counter.

Increment. Since the presence of the high bit indicates whether the counter is zero, the increment operation can just perform a fetch-and-add operation, and check whether the result has the high bit set. If so, it returns false.

Decrement. The decrement operation should decrement the reference count and return true if the reference count was brought to zero, or false otherwise. To decrement the counter, the

```
unsigned int zero = 1 << (b - 1);
unsigned int help = 1 << (b - 2);
unsigned int x;

bool increment_if_not_zero() {
  auto val = x.fetch_add(1);
  return (val & zero) == 0; }

bool decrement() {
  if (x.fetch_sub(1) == 1) {
    unsigned int e = 0;
    if (x.compare_exchange(e, zero)) return true;
    else if ((e & help) && (x.exchange(zero) & help)) return true;
  } return false; }

unsigned int load() {
  auto e = x.load();
  if (e == 0 && x.compare_exchange(e, zero | help)) return 0;
  return (e & zero) ? 0 : e; }
```

Figure 7.6: An implementation of a wait-free reference counter with constant time increment-if-not-zero, decrement, and load. Note that the `compare_exchange` operation, if unsuccessful, atomically loads the value of `x` into `e`.

algorithm uses a fetch-and-add and checks whether the counter hits zero. If it does, it must attempt to set the high bit to indicate this. This is done with a CAS. Note that if the CAS fails, it must be the case that an increment occurred that brought the counter back up from zero. In this case, the decrement can simply act as if the increment occurred before it, and hence report that it did not bring the counter to zero. A decrement that races with a load must handle one additional case described in the next paragraph.

Load. At first glace, the algorithm could try to just load the stored value, and return zero if the high bit is set. This however, is not necessarily correct if the stored value is zero. If the stored value is zero, the high bit might be about to be set, but an increment might race with it and bring the counter above zero. Reporting zero would therefore be incorrect. In order to achieve wait-freedom, the load operation therefore attempts to help set the high bit. If it successfully sets the high bit, it can return zero. If it fails, the unsuccessful CAS will return the current value of the counter.

If the load operation successfully helps to store the high bit, one of the decrements still needs to take responsibility for being the one who brought the counter to zero. To achieve this, the helping operation additionally writes the second-highest bit, to indicate to the decrement operation that it was helped. If a decrement operation fails to CAS the high bit but detects the helper bit, it can then perform a fetch-and-store (exchange in C++) to remove the helper bit. If it removes the helper bit, it takes credit for bringing the counter to zero.

7.4.4 Primitives for Weak Reference Counting

The addition of a weak reference count requires us to make changes to the use of the acquire-retire interface used behind our reference counting scheme. In the strong-only setting, a retired pointer always corresponds to a delayed decrement of the reference count. In the weak setting, our algorithm also needs to be able to delay decrements of the weak count.

Additionally, in the strong-only setting, obtaining a private pointer to a managed object meant that the strong reference count was at least one, and since the pointer through which it was obtained is protected, it is guaranteed to remain at least one. However, this property cannot be guaranteed for a *weak private pointer*, because a thread might be about to decrement the last remaining strong reference right as we acquire it. Therefore, to make weak private pointers safe, an additional round of deferral is required to defer the destruction of the managed object after its reference count reaches zero. This guarantees that after an acquire, if the strong reference count is at least one, the object will not be destroyed until after the protection of the private pointer is released. We refer to the destruction of the managed object as a *dispose* operation.

To facilitate these additional needs, instead of using a single instance of acquire-retire, our enhanced algorithm makes use of three instances—one for strong reference count decrements, one for weak decrements, and one for disposals.

Integrating these ideas, we extend the set of primitives for deferred reference counting with weak pointers as follows. Pseudocode is given in Figure 7.7. The **delayed_decrement**, **delayed_weak_decrement**, and **delayed_dispose** operations make use of three different instances of acquire-retire to delay a decrement to the strong or weak reference count, or the destruction of the managed object, until it is no longer protected by a corresponding acquire.

load_and_increment and **weak_load_and_increment** atomically load the value of the pointer stored at the given location and perform a safe increment of the strong or weak reference count respectively. Note that `load_and_increment` does not check whether the increment was successful, because these functions are only ever called on a pointer location that is storing a strong or weak reference respectively, and hence the reference count is already guaranteed to not be zero. It is a precondition violation to call this function on a pointer location that stores an object whose strong reference count is already zero.

increment and **weak_increment** attempt to increment the reference count or weak reference count respectively. The first returns true if successful. Note that `weak_increment` does not need to check for success because objects with a zero weak reference count are instantly destroyed, and hence it would be unsafe to attempt to increment the counter anyway. **decrement** decrements the strong reference count, and if it reaches zero, queues up a delayed *dispose*. A **dispose** destroys[2] the managed object and decrements the weak reference count. Similarly, **weak_decrement** decrements the weak reference count, and if it hits zero, immediately frees the managed object and its control data. Lastly, **expired** checks whether the managed object is still considered alive by checking that the reference count is not zero.

[2]We use destroy in the object-oriented sense to mean to recursively destroy all of its fields. If any of its fields are themselves reference-counted pointers, this would trigger their reference count decrements.

```
AcquireRetire<T> strongAR, weakAR, disposeAR;

void delayed_decrement(T* p) {
  strongAR.retire(p);
  auto x = strongAR.eject();
  decrement(x); }

void delayed_weak_decrement(T* p) {
  weakAR.retire(p);
  auto x = weakAR.eject();
  weak_decrement(x); }

void delayed_dispose(T* p) {
  disposeAR.retire(p);
  auto x = disposeAR.eject();
  dispose(x); }

T* load_and_increment(T** p) {
  auto ptr, guard = strongAR.acquire(p);
  if (ptr) increment(ptr);
  strongAR.release(guard);
  return ptr; }

T* weak_load_and_increment(T** p) {
  auto ptr, guard = weakAR.acquire(p);
  if (ptr) weak_increment(ptr);
  weakAR.release(guard);
  return ptr; }

bool increment(T* p) {
  return p->ref_cnt.increment_if_not_zero(); }

void weak_increment(T* p) {
  p->weak_cnt.increment_if_not_zero(); }

void decrement(T* p) {
  if (p->ref_cnt.decrement(1)) {
    delayed_dispose(p); } }

void dispose(T* p) {
  destroy(p->object);
  weak_decrement(p); }

void weak_decrement(T* p) {
  if (p->weak_cnt.decrement(1)) {
    delete p; } }

bool expired(T* p) {
  return p->ref_cnt.load() == 0; }
```

Figure 7.7: Primitives for implementing deferred reference counting with support for weak pointers.

7.4.5 Algorithms for Atomic Weak Pointers

Using the primitives from Figure 7.7, the algorithms for storing and loading to/from and CASing into an atomic weak pointer are very similar to those in CDRC. The main difference is that we must be careful to use the correct instance of acquire-retire for protection, and the correct kinds of increments/decrements. The algorithm that is most different from its strong counterpart is `load_private`. Pseudocode is given in Figure 7.8 and described below.

Storing a weak_ptr in an atomic_weak_ptr. This works the same as storing a `shared_ptr` in an `atomic_shared_ptr`. The algorithm first increments the weak reference count of `desired`, then uses a fetch-and-store (exchange in C++) to swap the managed object with the given one, and finally performs a delayed decrement of the weak reference count of the previously stored object.

Loading a weak_ptr from an atomic_weak_ptr. This is essentially the same as loading from an `atomic_shared_ptr`. The managed object is atomically loaded and has its weak reference count safely incremented, returning a `weak_ptr` to the managed object.

CASing into an atomic_weak_ptr. Compare and swap begins by protecting the pointer owned by `desired`. If the CAS is successful, it increments the weak reference count of desired and performs a delayed decrement of the weak reference count of `expected`. Note that the guard must be acquired before performing the CAS because otherwise, the CAS might succeed while another process clobbers `desired`, destroying it before the reference count increment happens.

Creating a private pointer from an atomic_weak_ptr. Creating a private pointer from an `atomic_weak_ptr` is slightly more complicated than taking one from an `atomic_shared_ptr`. The main idea is to try to acquire a protected pointer to the managed object that prevents the object from being disposed, and, if the managed object has not expired (the strong reference count is at least one), return a private pointer containing the protected pointer. If the try_acquire fails, the backup plan is to attempt to increment the reference count[3]. In case the managed object has already been disposed before protecting the pointer, the algorithm first acquires protection against a possible weak decrement, since, otherwise, the control data could be deleted mid-operation.

If the strong reference count is zero, the obvious algorithm would just return a private pointer containing a null pointer. However, this strategy would result in the operation not being linearizable, because the reference count could be in the process of being decremented right as the pointer is acquired. This would allow for situations where the `atomic_weak_ptr` always points to a live object, but the `load_private` may return null if the object was replaced in between the acquire and the read of the reference count. Therefore, if the reference count is zero, the algorithm only returns a null pointer if the `atomic_weak_ptr` still manages the same acquired pointer. If not, the algorithm retries from the beginning. This retrying causes `load_private` to be lock-free but not wait-free.

[3]This only happens with the hazard pointer implementation if too many private pointers are held at once such that the announcement array runs out of slots. EBR, IBR and Hyaline never fail.

```
void atomic_weak_ptr<T>::store(const weak_ptr<T>& desired) {
  if (desired.ptr) weak_increment(desired.ptr);
  auto old_ptr = this->ptr.exchange(desired.ptr);
  if (old_ptr) delayed_weak_decrement(old_ptr); }

weak_ptr<T> atomic_weak_ptr<T>::load() {
  auto ptr = weak_load_and_increment(addressof(this->ptr));
  return weak_ptr(ptr); }

bool atomic_weak_ptr<T>::compare_and_swap(
  const weak_ptr<T>& expected, const weak_ptr<T>& desired) {
  auto ptr, guard = weakAR.acquire(addressof(desired.ptr));
  if (compare_and_swap(this->ptr, expected.ptr, ptr)) {
    if (ptr) weak_increment(ptr);
    if (expected.ptr) delayed_weak_decrement(expected.ptr);
    weakAR.release(guard);
    return true; }
  else {
    weakAR.release(guard);
    return false; } }

weak_private_ptr<T> atomic_weak_ptr<T>::load_private() {
  while (true) {
    auto ptr, weak_guard = weakAR.acquire(addressof(this->ptr));
    auto _, dispose_guard=disposeAR.try_acquire(addressof(ptr));
    if (dispose_guard == ⊥ && ptr) increment(ptr);
    if (ptr && !expired(ptr)) {
      weakAR.release(weak_guard);
      return weak_private_ptr(ptr, dispose_guard); }
    else {
      disposeAR.release(dispose_guard);
      weakAR.release(weak_guard);
      if (ptr == null || this->ptr == ptr)
        return weak_private_ptr(null); } }

void weak_private_ptr<T>::release() {
  if (this->guard != ⊥) disposeAR.release(this->guard);
  else decrement(this->ptr); }
```

Figure 7.8: C++-like pseudo-code for atomic weak pointers.

```
 1 class doubly_linked_queue<V> {
 2   struct Node {
 3     V value;
 4     atomic_shared_ptr<Node> next;
 5     atomic_weak_ptr<Node> prev;
 6     Node(V v) { value = v; next = null; prev = null; } };

 8   atomic_shared_ptr<Node> head, tail;

10   void enqueue(V v) {
11     shared_ptr<Node> new_node = shared_ptr<Node>::make_shared(v);
12     critical_section_guard guard;
13     while (true) {
14       private_ptr<Node> ltail = tail.load_private();
15       new_node->prev.store(ltail);
16       // Help the previous enqueue set its next ptr
17       weak_private_ptr<Node> lprev = ltail->prev.load_private();
18       if (lprev && lprev->next == null) lprev->next.store(ltail);
19       if (tail.compare_and_swap(ltail, new_node)) {
20         ltail->next.store(std::move(new_node));
21         return; } } }

23   std::optional<V> dequeue() {
24     critical_section_guard guard;
25     while (true) {
26       private_ptr<Node> lhead = head.load_private();
27       private_ptr<Node> lnext = lhead->next.load_private();
28       if (!lnext) return {}; // Queue is empty
29       if (head.compare_and_swap(lhead, lnext)) {
30         return {lnext->value}; } } } };
```

Figure 7.9: Ramalhete and Correia's concurrent doubly-linked queue [141] implemented using our weak pointer interface (C++-like pseudocode).

7.4.6 Example Usage

An example of how to apply our `weak_ptr` interface to Ramalhete and Correia's doubly-linked queue [141] is shown in Figure 7.9. The `prev` pointer of each node is stored in an atomic weak pointer, whereas the `next` pointers are stored in atomic shared pointers. The `critical_section_guard` (on lines 12 and 24) is only needed if generalized acquire-retire was implemented from a protected-region SMR technique. The `critical_section_guard` is responsible for calling `begin_critical_section` in its constructor and also `end_critical_section` in its destructor.

7.5 Experimental Evaluation

We implemented our techniques as a C++ library[4] and evaluated them on a series of benchmarks. Our experiments were run on a 4-socket 72-core machine (4× Intel(R) Xeon(R) E7-8867

[4]Available at `https://github.com/cmuparlay/concurrent_deferred_rc`

v4, 2.4GHz) with 2-way hyperthreading, a 45MB L3 cache, and 1TB of main memory. Memory was interleaved across sockets using *numactl -i all*, and we used the *jemalloc* allocator [63]. Experiments were written in C++ and compiled with GCC 9.2.1 with O3 optimization. Our experiments vary the number of threads from 1 to 192, which allows us to measure the effect of oversubscription, as our hardware supports 144 threads.

7.5.1 Comparing Manual and Automatic Techniques

We applied the approach in Section 7.3 to three different manual SMR techniques, EBR [71], IBR (more specifically, 2GEIBR) [167], and Hyaline (more specifically, Hyaline-1) [127], to construct three new concurrent reference counting implementations, which we call RCEBR, RCIBR, and RCHyaline, respectively. The goal of this section is to understand the overhead of making manual techniques automatic as well as to compare the performance of RCEBR, RCIBR, and RCHyaline with the fastest existing reference counting algorithm. The two fastest existing reference counting algorithms that we are aware of are FRC [157] and CDRC. We chose to compare with CDRC because FRC does not support marked pointers which are required in all of our benchmarks. For consistency, we rename CDRC to RCHP in the graphs as it is a combination of hazard-pointers and reference counting.

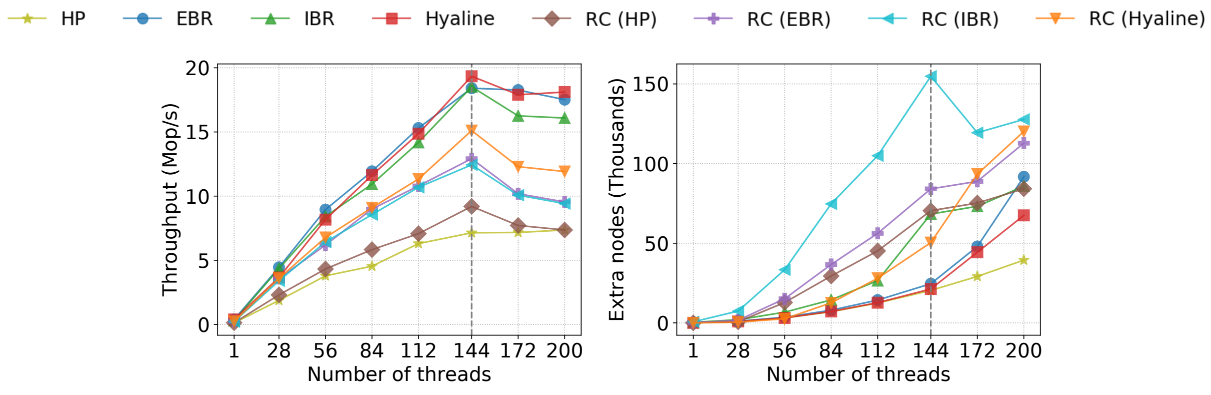
As for manual techniques, we compare with HP, EBR, IBR, and Hyaline. An important parameter to tune when using EBR and IBR is how often the global epoch gets incremented. Incrementing too often could bottleneck scalability whereas incrementing infrequently would increase memory usage. For EBR and RCEBR, we found a good rate to be one increment every 10 allocations and for IBR and RCIBR, we found this to be one increment every 40 allocations.

For both HP and RCHP, we found that prefetching appropriately significantly increased throughput. In particular, before announcing a pointer in the hazard array, we prefetch the cache line that it points to because there is a good chance we will follow the pointer after succeeding in announcing it. The benefit of this is that we can start loading the cache line before the memory barrier, which is an expensive operation. Note that due to this prefetching optimization, our throughput reported here for HP and RCHP is greater than the throughput of the same schemes in Chapter 6.

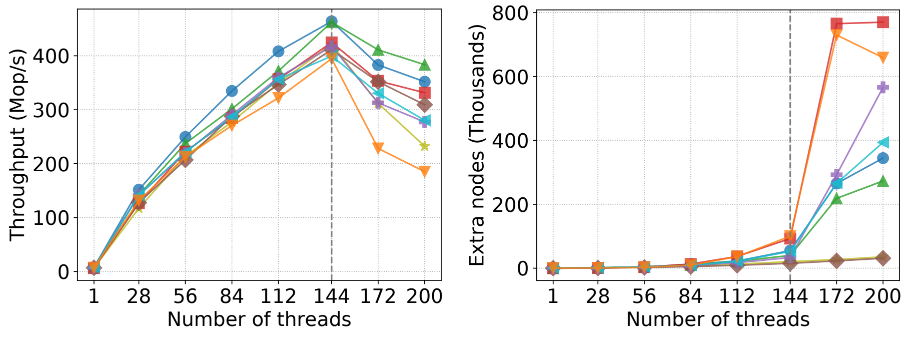
To benchmark performance, we applied these memory reclamation techniques to three different lock-free data structures: Harris-Michael list [80, 116], Michael hash table [116], and Natarajan-Mittal tree [122].

Chapter 6 explains why HP and IBR are not safe to use with the Natarajan-Mittal tree directly. This is essentially because traversals in the Natarajan-Mittal tree can continue through marked nodes. We still include these numbers in our experiments for reference, even though these experiments occasionally crash. Modifying the Natarajan-Mittal tree to work with HP and IBR would likely make it slower. Note that an advantage of RCHP and RCIBR is that they work with Natarajan-Mittal tree without any such modifications.

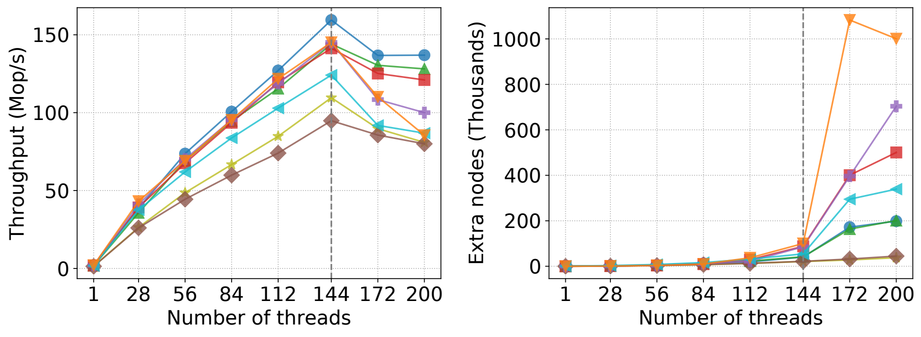
Range query workload. We begin by analyzing the experiment shown in Figure 7.12. In this workload, we initialized the Natarajan-Mittal tree with 100K keys randomly selected from the key range $[0, 200K)$, and then performed update operations (half insert, half delete) and range queries. We use a sequential range query algorithm, which is not linearizable. with equal



(a) List. N=1000, updates=10%. Throughput (L), Memory (R)

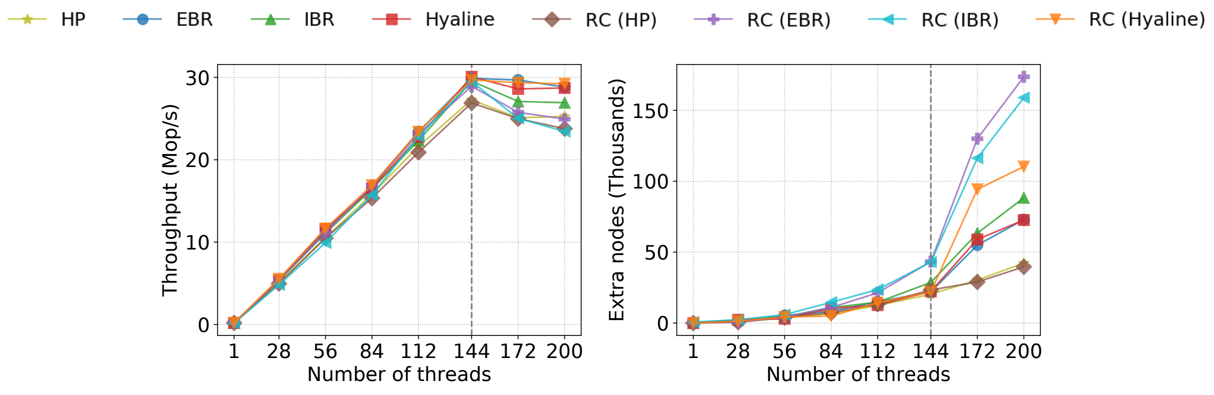


(b) Hash table. N=100K, updates=10%. Throughput (L), Memory (R)

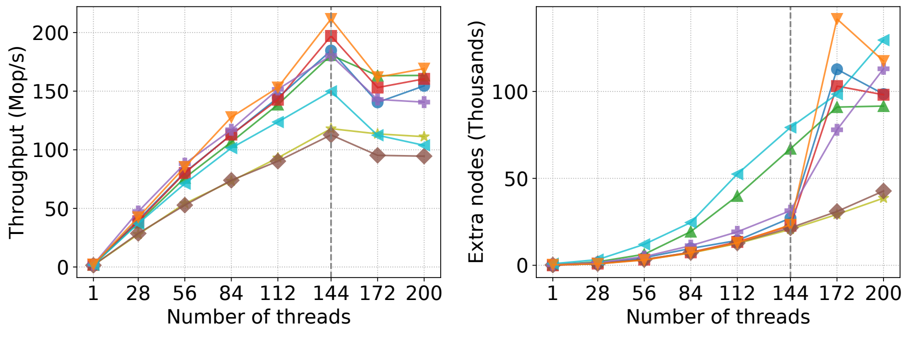


(c) BST. N=100K, updates=10%. Throughput (L), Memory (R)

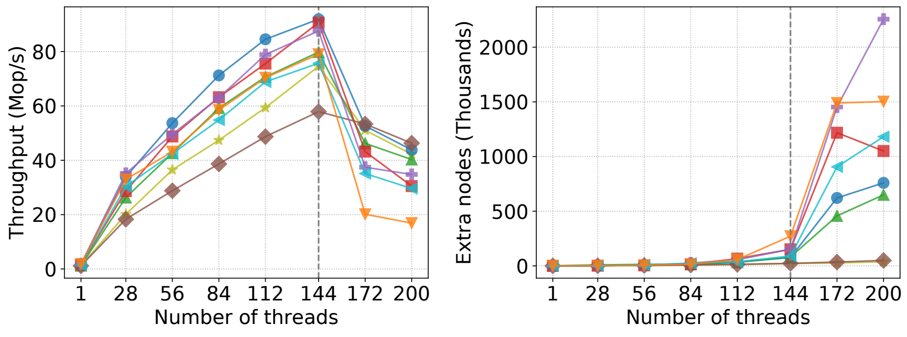
Figure 7.10: Benchmark comparing manual and automatic SMR techniques. Figure 7.10(a) shows results for a Harris-Michael list, Figure 7.10(b) for a Michael hash table, and Figure 7.10(c) for a Natarajan-Mittal tree.



(a) BST. N=100M, updates=10%. Throughput (L), Memory (R)



(b) BST. N=100K, updates=1%. Throughput (L), Memory (R)



(c) BST. N=100K, updates=50%. Throughput (L), Memory (R)

Figure 7.11: Benchmark comparing manual and automatic SMR techniques on a Natarajan-Mittal tree.

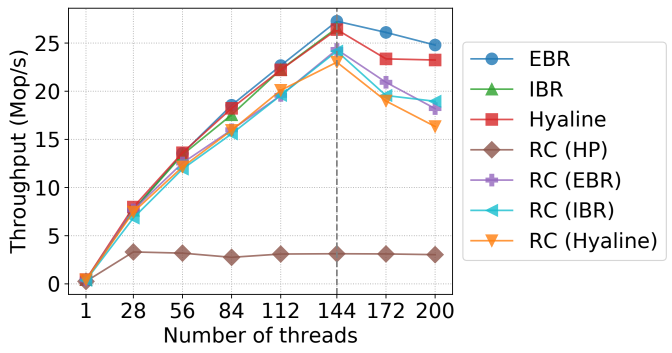


Figure 7.12: Natarajan-Mittal tree - Range query experiments: 50% updates, 50% range queries of size 64.

probability. Each update operation selects a uniform random key from $[0, 200K)$ to insert/delete and each range query selects a uniform random key k from the same range and queries for all keys in the interval $[k, k+64)$. In this experiment, we found that RCEBR, RCIBR, and RCHyaline outperform RCHP by more than 7x on 144 threads. This is because during a range query, the entire path from the current node to the root needs to be protected by `private_ptrs`, so RCHP eventually runs out of announcement locations and starts relying on reference count increments, which is significantly more expensive. RCEBR, RCIBR, and RCHyaline also performs similarly to their manual counterparts, performing within 10-15% at 144 threads.

Other workloads. Figure 7.10 shows the throughput and memory usage of these SMR technique on a wide variety of workloads. These workloads only contain updates and single point lookups. For example, Figure 7.10(c) shows a workload where the Natarajan-Mittal tree is initialized with 100K keys, and each process performs 10% update operations and 90% lookups. Again, all keys are chosen uniformly randomly from a key range twice the initial size of the data structure. For the hash table experiments, we initialized the number of buckets so that the average load factor is 1.

When update frequency is low (Figure 7.11(b)), RCEBR has almost the exact same throughput as EBR and RCHyaline is actually slightly faster than Hyaline. However, RCIBR ends up being about 20% slower than IBR and this overhead comes from two main factors. First, RCIBR adds both a reference count and a birth epoch to each node, and this increase in size accounts for about half of the performance difference. Second, each `try_acquire` in RCIBR requires reading a thread local variable storing the process id and this access is surprisingly slow, accounting for the other half of the performance difference. Overall, on the BST experiments with 144 threads, RCEBR performs within 10% of EBR (in terms of throughput) and RCHyaline performs within 15% of Hyaline. Also, RCEBR is up to 1.7x faster than RCHP in Figure 7.10(c).

In the non-oversubscribed scenarios, the automatic version of each memory reclamation scheme tends to use a similar amount of memory to the manual version. However in the linked list experiment and also in oversubscribed cases, the automatic version tends to have several times more memory overhead. This is because in reference counting techniques, each retired pointer could recursively prevent the collection of many nodes beyond the one it directly points to.

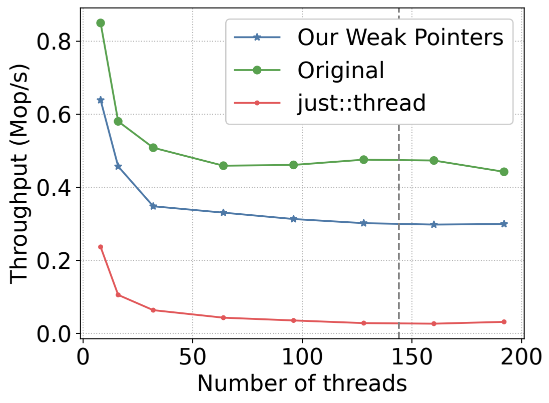


Figure 7.13: Benchmark results for atomic weak pointers. Original is the optimized doubly linked queue of Ramalhete and Correia [141] that uses a custom manual memory management technique. Our algorithm uses atomic weak pointers powered by the hazard pointer implementation of acquire-retire. `just::thread` is a commercial library of atomic shared and weak pointers.

7.5.2 Evaluation of Atomic Weak Pointers

We compare our implementation of atomic weak pointers with the best known existing lock-free implementation, the just::thread library [169], and against a manually memory-managed data structure. For our comparison we use the doubly linked queue of Ramalhete and Correia [141]. This queue is a good candidate since it uses back pointers that can be represented using weak pointers. For this comparison, we use our reference counting library powered by the hazard pointer implementation of acquire-retire. We found that the main bottleneck of the throughput of the data structure is the contention on the CAS operations, and hence the different choices of acquire-retire implementation only made minor differences to the performance.

The original implementation of the data structure does not use a general purpose memory management scheme, but actually uses a customized version of hazard pointers specifically engineered for it. This modified hazard pointers scheme allows announced nodes to protect not only themselves, but also the nodes adjacent to them. This reduces the number of memory barriers required by the algorithm. For this reason, it is not likely that a general purpose memory management scheme would outperform it.

In our experiment, we initialize a single queue with P elements, and have P threads. Each thread repeatedly pops an element from the queue and then reinserts it. We then measure the number of such operations that were performed per second. Each benchmark is repeated five times for stability. The results of this experiment are depicted in Figure 7.13.

The biggest difference in performance occurs at $P = 1$ (not depicted on the plot due to scale), where the original implementation is 4.5x faster than our weak pointers, and 67x faster than just::thread. At $P = 8$ threads, our weak pointer implementation is just 19% slower than the manual approach, and 4.2x faster than just::thread. This trend roughly continues to $P = 192$, where our weak pointers are 33% slower than the manual approach, but 10x faster than just::thread. Given that the original implementation uses a memory management approach that is customized to the data structure at hand, these results are very promising for a completely automatic approach. Furthermore, we substantially outperform the best existing automatic approach at all thread counts.

7.6 Conclusion

In this work, we showed that an automatic memory reclamation technique can compete with the best manual techniques, and showed that such a technique can also support atomic weak pointers. Though perhaps it is not yet time to completely retire manual memory reclamation, we believe that these results show, even more strongly than previous results, that we are getting close, and that automatic memory management should be preferable in a majority of situations.

Part IV

Conclusion

Chapter 8

Conclusion

The thesis statement was that *general techniques, along with appropriate abstractions and library implementations, can greatly simplify the design and implementation of efficient concurrent algorithms.* Breaking this down, this statement essentially consists of three parts: efficiency, ease-of-use, and generality. We show how the thesis supports each of these points separately.

- **Efficiency**: In terms of practical performance, the general techniques we develop are competitive with the best hand-optimized solutions. In Part I, we showed that lock-free locks add very little overhead relative to traditional blocking locks and perform significantly better in cases where threads may be paused, for example due to oversubscription. In Part II, we showed that our snapshotting approach outperforms existing state-of-the-art solutions. For example, applying versioned pointers to a simple B-tree resulted in significantly faster range queries than specialized data structures designed specifically for range queries. In Part III, we showed that our automatic safe memory reclamation schemes are competitive with manual ones in terms of both time and space usage. On the theoretical side, we proved worst-case time bounds for our versioned CAS and reference counting algorithm in Chapters 4 and 6, respectively.
- **Ease of use**: We provide an easy-to-use library interface for our all of our techniques. Our lock-free locks technique allows programmers to write lock-free code using just the familiar interface of locks, avoiding many of the intricacies and subtleties of lock-free programming. Our versioned pointer and versioned CAS approaches provide an easy way to take snapshots of concurrent memory that just involves replacing pointers or CAS objects with these new types. They also allow multi-point queries to be implemented by simply running standard sequential algorithms on a snapshot of memory. Finally, our automatic memory reclamation techniques avoid many of the usability difficulties and common pitfalls in manual memory reclamation. Using our library just involves replacing raw pointers with the new pointer types we provide.
- **Generality**: All of our techniques are applicable to a wide range of data structures. Lock-based data structures are very common and our lock-free locks approach works for any lock-based data structure without deadlocks or livelocks. Our snapshotting approach works for any data structure where mutable state is stored in CAS objects or pointer

types. It also supports a wide range of multi-point queries whereas most previous work has focused on range queries in particular. Our automatic memory reclamation schemes works as long as reference cycles are broken before they become unreachable. The widely used shared pointer interface from C++ has the same requirement and most concurrent data structures in the literature satisfy it.

Throughout this thesis, we developed new algorithms, new abstractions and new library implementations. All of these advancements were important for showing the simplicity, efficiency, and wide applicability of our techniques. We hope that they will help simplify the design of future concurrent algorithms and data structures, and make concurrent programming easier and more accessible to a wider audience.

Bibliography

[1] Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. *ACM SIGPLAN Notices*, 52(8):75–88, 2017. 105

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993. 5, 47

[3] Archita Agarwal, Zhiyu Liu, Eli Rosenthal, and Vikram Saraph. Linearizable iterators for concurrent data structures. *CoRR*, abs/1705.08885, 2017. URL `http://arxiv.org/abs/1705.08885`. 5, 95

[4] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2021. doi: 10.1145/3453483.3454060. 9

[5] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Turning manual concurrent memory reclamation into automatic reference counting. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022. doi: 10.1145/3519939.3523730. 9

[6] James H. Anderson. Multi-writer composite registers. *Distributed Comput.*, 7(4):175–195, 1994. 5

[7] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 14–27, 2018. 5, 7, 49, 60, 66, 86, 88, 96

[8] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *USENIX Annual Technical Conference*, 2018. 3, 34, 37

[9] Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems (TOCS)*, 52(4):729–762, 2013. 19

[10] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 336–343, 2008. 47, 95

[11] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 299–308, 2013. 95

[12] David F Bacon, Clement R Attanasio, Han B Lee, VT Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM*

Conference on Programming Language Design and Implementation (PLDI), 2001. 102, 105

[13] Henry G Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.*, 29(9):38–43, 1994. 102

[14] Greg Barnes. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 261–270, 1993. 17, 21, 27, 41

[15] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. *ACM Transactions on Parallel Computing (TOPC)*, 7(3), June 2020. 49, 60, 66, 95

[16] R. Bayer and M. Schkolnick. *Concurrency of Operations on B-Trees.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0934613656. 4, 31

[17] 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. 18, 21, 27, 41

[18] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019. 9, 21, 27, 41, 81

[19] Naama Ben-David, Guy E Blelloch, Yihan Sun, and Yuanhao Wei. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019. 5, 9, 96

[20] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection. In *International Symposium on Distributed Computing (DISC)*, 2021. doi: 10.4230/LIPIcs.DISC.2021.12. 9, 17, 19

[21] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022. 9, 32

[22] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. The flock library. `https://github.com/cmuparlay/flock`, 2022. 88

[23] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. Lock-free locks revisited, 2022. 9

[24] Naama Ben-David, Michal Friedman, and Yuanhao Wei. Brief announcement: Survey of persistent memory correctness conditions. In *International Symposium on Distributed Computing (DISC)*, 2022. doi: 10.4230/LIPIcs.DISC.2022.41. 9

[25] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, December 1983. 5, 48, 49, 96

[26] Stephen M Blackburn and Kathryn S McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003. 102, 105

[27] Guy E. Blelloch and Yuanhao Wei. Brief announcement: Concurrent fixed-size allocation and free in constant time. In *International Symposium on Distributed Computing (DISC)*, 2020. doi: 10.4230/LIPIcs.DISC.2020.51. 9

[28] Guy E Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In *International Symposium on Distributed Computing (DISC)*, 2020. 9, 108, 117, 118

[29] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016. 60

[30] Guy E. Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 27, 41

[31] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020. URL `https://cmuparlay.github.io/parlaylib/`. 34

[32] Alex Brodsky and Faith Ellen Fich. Efficient synchronous snapshots. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 70–79, 2004. 5

[33] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2010. 4, 34, 35, 49, 60, 66, 95

[34] Jeremy Brown, J. P. Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 248–257, 2002. 18, 41

[35] Trevor Brown. Java lock-free data structure library. Available from `https://bitbucket.org/trbot86/implementations/src/master/java`. 66

[36] Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *Conf. on Principles of Distributed Systems (OPODIS)*, volume 7702, pages 31–45, 2012. 5, 60, 66, 95

[37] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *ACM Symposium on Principles of Distributed Computing (PODC)*, page 13–22, 2013. 4, 17, 37, 41

[38] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 329–342, 2014. 6, 7, 35, 48, 49, 60, 61, 66

[39] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 261–270, 2015. 64, 101, 104

[40] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006. 96

[41] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, 2015. 41

[42] Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *IEEE International Conference on Distributed Computing and Networking (ICDCN)*, pages 9:1–9:10, 2017. 5, 96

[43] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010. 35, 92

[44] Intel Corp. Intel® 64 and ia-32 architectures developer's manual: Vol. 3b, section 18.17, time-stamp counter, December 2022. 73

[45] Andreia Correia and Pedro Ramalhete. Strong trylocks for reader-writer locks. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018. 139

[46] Andreia Correia, Pedro Ramalhete, and Pascal Felber. OrcGC: automatic lock-free memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2021. 8, 102, 103, 105, 121, 128

[47] Cplusplus.com. Documentation for std::find_if. `http://www.cplusplus.com/reference/algorithm/find_if/` accessed 31 December, 2020. 66

[48] 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. 3, 4, 34, 38, 74, 88, 128

[49] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013. 18, 41

[50] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012. 18, 41

[51] David Detlefs, Paul Alan Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002. 8, 101, 102, 104, 112

[52] L Peter Deutsch and Daniel G Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, 1976. 102, 105

[53] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013. 5, 96

[54] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *International Symposium on Memory Management (ISMM)*, pages 163–174, 2002. 17

[55] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *International Symposium on Distributed Computing (DISC)*, pages 194–208. Springer, 2006. 87

[56] Thomas Dickerson. Adapting persistent data structures for concurrency and speculation, 2020. 5

[57] Nuno Diegues and Paolo Romano. Time-warp: Efficient abort reduction in transactional memory. *ACM Transactions on Parallel Computing (TOPC)*, 2(2), June 2015. 96

[58] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2014. 4, 35

[59] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Computer and System Sciences*, 38(1):86–124, 1989. 5

[60] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *J. ACM*, 41(5):943–959, September 1994. 5

[61] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 13–22, 2007. 105

[62] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2010. 1, 4, 6, 17, 35, 48, 49, 60, 61, 63, 84

[63] J. Evans. *Scalable memory allocation using jemalloc*, 2019 (accessed November 5, 2019). `https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`. 121, 146

[64] Facebook. *Facebook Open Source Library*, 2020 (accessed June 5, 2020). `https://github.com/facebook/folly`. 8, 101, 102, 104, 110, 111, 121

[65] Panagiota Fatourou and Nikolaos D. Kallimanis. Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using cas. In *ACM Symposium on Principles of Distributed Computing (PODC)*, page 33–42, 2007. 5

[66] Panagiota Fatourou, Yiannis Nikolakopoulos, and Marina Papatriantafilou. Linearizable wait-free iteration operations in shared double-ended queues. *Parallel Processing Letters*, 27(2):1–17, 2017. 5, 95

[67] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019. 5, 49, 66, 68, 86, 95

[68] 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. 17, 42

[69] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, page 179–188, 2011. 5, 48, 49, 96

[70] Faith Ellen Fich. How hard is it to take a snapshot? In *Proc. 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 3381 of *LNCS*, pages 28–37, 2005. 5, 95

[71] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004. 32, 49, 64, 101, 103, 124, 146

[72] Keir Fraser and Tim Harris. Concurrent programming without locks. *Theory of Computing Systems (TOCS)*, 25(2):5–es, 2007. 17, 86

[73] 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 *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2020. 9, 128

[74] Anders Gidenstam and Marina Papatriantafilou. Lfthreads: A lock-free thread library. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2007. 42

[75] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8), 2009. 102, 106

[76] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 2011. 60

[77] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 260–269, 2002. 19

[78] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In *International Symposium on Distributed Computing (DISC)*, 2020. 17, 42

[79] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008. ISSN 0018-8670. 32, 101, 103

[80] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*, pages 300–314. Springer, 2001. 4, 6, 38, 41, 48, 60, 63, 84, 112, 124, 146

[81] 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, 2002. 4, 17, 42

[82] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007. 7, 8, 101

[83] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2006. 3, 4, 34

[84] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1), 1991. 41

[85] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012. 34

[86] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *International Symposium on Distributed Computing (DISC)*, 2002. doi: 10.1007/3-540-36108-1_23. 106

[87] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2), May 2005. doi: 10.1145/1062247.1062249. 8, 101, 102, 103, 104, 106, 112, 117, 121

[88] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990. 12, 59, 108

[89] Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1–12, 2012. 47, 95

[90] Peter Zilahy Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961. 21

[91] Prasad Jayanti. f-arrays: Implementation and applications. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279, 2002. doi: 10.1145/571825.571875. 95

[92] Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *ACM Symposium on Theory of Computing (STOC)*, page 723–732, 2005. 5

[93] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795. 7

[94] Nikolaos D. Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2015. 95

[95] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(5):942–965, sep 2006. 41

[96] Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski. Jiffy: A lock-free skip list with batch updates and snapshots. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022. 73, 87, 88, 96

[97] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, 2012. doi: 10.1145/2370036.2145835. 121

[98] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *IEEE International Conference on Distributed Computing and Networking (ICDCN)*, pages 212–226, 2014. 5, 48, 49, 96

[99] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3), 1980. 4, 19, 34

[100] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981. 19, 22

[101] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. 96

[102] Hyonho Lee. Fast local-spin abortable mutual exclusion with bounded space. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2010. 8, 101, 102, 104

[103] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *IEEE International Conference on Data Engineering (ICDE)*, 2013. 34, 88

[104] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *International Workshop on Data Management on New Hardware (DaMoN)*, 2016. 4, 5, 34, 74, 88

[105] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2001. 102, 105

[106] The GNU C++ Library. *The GNU C++ Library*, 2019 (accessed November 5, 2019). `https://gcc.gnu.org/onlinedocs/libstdc++/`. 121, 130, 139

[107] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 21–35, 2017. 96

[108] linux. Documentation for futex. `https://linux.die.net/man/2/futex/` accessed 26 April, 2023. 35

[109] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015. 41

[110] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems (EuroSys)*, 2012. 4

[111] Paul E. McKenney and Jonathan Walpole. Introducing technology into the linux kernel: A case study. *SIGOPS Oper. Syst. Rev.*, 42(5), July 2008. doi: 10.1145/1400097.1400099. URL `https://doi.org/10.1145/1400097.1400099`. 4

[112] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), February 1991. 35

[113] Alisdair Meredith. Revising atomic_shared_ptr for C++20, 2017. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0718r2.html`. 104

[114] Ian Mertz, Toniann Pitassi, and Yuanhao Wei. Short proofs are hard to find. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*, 2019. doi: 10.4230/LIPIcs.ICALP.2019.84. 9

[115] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical report, Computer Science Department, University of Rochester, 1995. 8, 101, 102

[116] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2002. 124, 146

[117] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004. 101, 103, 106, 111, 117, 124

[118] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996. 6, 41, 48, 60

[119] Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. Proposed wording for concurrent data structures: Hazard pointer and read-copy-update (RCU), 2017. `http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0566r3.pdf`. 109, 110

[120] Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. Hazard pointers: Proposed interface and wording for concurrency TS 2, 2019. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1121r1.pdf`. 106

[121] Microsoft. Microsoft's c++ standard library, 2021 (accessed November 17, 2021). `https://github.com/microsoft/STL`. 137, 139

[122] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2014. 4, 35, 41, 84, 108, 112, 124, 127, 146

[123] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, page 368–384, 2022. 86, 87, 88, 96

[124] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015. 5, 48, 49, 86, 96

[125] Ruslan Nikolaev and Binoy Ravindran. Universal wait-free memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2020. 102, 104, 128

[126] Ruslan Nikolaev and Binoy Ravindran. Brief announcement: Crystalline: Fast and memory efficient wait-free reclamation. In *International Symposium on Distributed Computing (DISC)*, 2021. 104

[127] Ruslan Nikolaev and Binoy Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 987–1002, 2021. 104, 146

[128] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occassion of his*

60th Birthday, pages 358–369. Springer, 2015. 95

[129] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 239–248, 2015. 5, 95

[130] Oracle. Java weakly consistent iterators. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#Weakly` accessed 31 December 2020. 5

[131] Christos H Papadimitriou and Paris C Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems (TODS)*, 9(1):89–99, 1984. 5, 96

[132] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 16–25, 2010. 96

[133] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: Selective multi-versioning stm. In *International Symposium on Distributed Computing (DISC)*, pages 125–140, 2011. 96

[134] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *International Symposium on Distributed Computing (DISC)*, pages 224–238, 2013. doi: 10.1007/978-3-642-41527-2_16. 5, 95

[135] Dan Plyukhin. Distributed reference counting for asynchronous shared memory, 2015 (accessed November 5, 2019). `http://rucs.ca/theory-of-computation/distributed-reference-counting-for-asynchronous-shared-memory`. 8, 101, 102, 104

[136] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment (PVLDB)*, 5(12), 2012. 5, 96

[137] Aleksandar Prokopec. Snapqueue: Lock-free queue with constant time snapshots. In *ACM SIGPLAN Symposium on Scala*, pages 1–12, 2015. 95

[138] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 151–160, 2012. doi: 10.1145/2145816.2145836. 95

[139] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33 (6), 1990. 4

[140] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, 2002. 42

[141] Pedro Ramalhete and Andreia Correia. Doublelink - a low-overhead lock-free queue. `http://concurrencyfreaks.blogspot.com/2017/01/doublelink-low-overhead-lock-free-queue.html`. 130, 145, 150

[142] Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras-non-blocking memory reclamation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017. 101, 104, 124

[143] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, 2019. 17

[144] D. Reed. Naming and synchronization in a decentralized computer system. Technical Report LCS/TR-205, EECS Dept., MIT, September 1978. 5, 48, 49, 96

[145] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing (DISC)*, pages 284–298. Springer, 2006. 96

[146] Wenjia Ruan, Yujie Liu, and Michael Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. *ACM Trans. Archit. Code Optim.*, 10(4), dec 2013. 73, 96

[147] Neil Sarnak and Robert E Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986. 5, 48

[148] Niloufar Shafiei. Non-blocking doubly-linked lists with good amortized complexity. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2016. 17, 19

[149] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10 (2):99–116, 1997. 17

[150] Gali Sheffi, Pedro Ramalhete, and Erez Petrank. EEMARQ: efficient lock-free range queries with memory reclamation. *CoRR*, abs/2210.17086, 2022. doi: 10.48550/arXiv.2210.17086. URL `https://doi.org/10.48550/arXiv.2210.17086`. 96

[151] Anubhav Srivastava and Trevor Brown. Elimination (a, b)-trees with fast, durable updates. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 416–430, 2022. 38, 88

[152] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. PAM: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018. 60

[153] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)*, 13(2), 2019. 96

[154] Håkan Sundell. Wait-free reference counting and memory management. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005. 8, 101, 102, 104

[155] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, 2008. 19

[156] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–368, 2014. 41

[157] Charles Tripp, David Hyde, and Benjamin Grossman-Ponemon. Frc: a high-performance concurrent parallel deferred reference counter for c++. *Acm Sigplan Notices*, 53(5):14–28, 2018. 105, 146

[158] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 18–32, 2013. 96

[159] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Principles of Database Systems (PODS)*, pages 212–222, 1992. 17, 21, 27, 32, 41

[160] John D. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995. 8, 101, 102

[161] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *IEEE International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018. 17

[162] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. Building a bw-tree takes more than just buzz words. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 05 2018. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196895. 3, 4

[163] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures, 2020. 9

[164] 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 Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2021. 9, 60

[165] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. Flit: A library for simple and efficient persistent algorithms. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2022. 9

[166] Yuanhao Wei, Guy E. Blelloch, Panagiota Fatourou, and Eric Ruppert. Practically and theoretically efficient garbage collection for multiversioning. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2023. doi: 10.1145/3572848.3577508. 9

[167] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *SIGPLAN Not.*, 53(1):1–13, 2018. 101, 102, 104, 124, 128, 133, 134, 146

[168] Anthony Williams. *C++ concurrency in action: practical multithreading*. Manning Publ., 2012. 102, 104, 121

[169] Anthony Williams. *just::thread Concurrency Library*, 2019 (accessed November 5, 2019). `https://www.stdthread.co.uk`. 8, 101, 102, 104, 111, 121, 130, 137, 150

[170] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. *ACM Transactions on Parallel Computing (TOPC)*, 8(2):1–38, 2021. 17, 49, 60, 66, 68, 88, 95

[171] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*

(PVLDB), 10, 2017. 5, 48, 49, 86, 96

[172] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 1629–1642, 2016. 96