1-1-2012

Extending Relativistic Programming to Multiple Writers

Philip William Howard
Portland State University

Extending Relativistic Programming to Multiple Writers

by

Philip William Howard

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Jonathan Walpole, Chair
Robert Daasch
James Hook
Mark Jones
Paul E. McKenney
Bryant York

Portland State University
2012

Abstract

For software to take advantage of modern multicore processors, it must be safely concurrent and it must scale. Many techniques that allow safe concurrency do so at the expense of scalability. Coarse grain locking allows multiple threads to access common data safely, but not at the same time. Non-Blocking Synchronization and Transactional Memory techniques optimistically allow concurrency, but only for disjoint accesses and only at a high performance cost. Relativistic programming is a technique that allows low overhead readers and joint access parallelism between readers and writers. Most of the work on relativistic programming has assumed a single writer at a time (or, in partitionable data structures, a single writer per partition), and single writer solutions cannot scale on the write side.

This dissertation extends prior work on relativistic programming in the following ways:

1. It analyses the ordering requirements of lock-based and relativistic programs in order to clarify the differences in their correctness and performance characteristics, and to define precisely the behavior required of the relativistic programming primitives.

2. It shows how relativistic programming can be used to construct efficient, scalable algorithms for complex data structures whose update operations involve multiple writes to multiple nodes.

3. It shows how disjoint access parallelism can be supported for relativistic writers, using Software Transactional Memory, while still allowing low-overhead, linearly-scalable, relativistic reads.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0719851.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

CPU manufacturers have transitioned from supplying faster processors to supplying processors that can execute more threads simultaneously [32]. This change has meant a change in software: software must be *concurrent* in order to take advantage of modern multicore hardware. In order to be correctly concurrent, software must preserve data-safety in the presence of concurrent access, and it must scale so as more processors are available, more work will get done. Both of these properties should be considered part of the correctness criteria for a concurrent program. If a program produces incorrect results because of data races, then clearly it is incorrect. But if a program does not scale, a non-concurrent solution is both simpler and at least as efficient.

When multiple threads are accessing common data, there is the potential for an interleaved execution that leads to incorrect results. Consider a counter that is to be incremented by two threads. If both threads read the counter, then both increment the value, then both write the result back to the counter, the end result is that one of the updates was lost. Some form of synchronization is required to avoid this incorrect behavior. Unfortunately, adding synchronization to a program not only adds to its complexity, it also tends to negatively impact both performance and scalability.

Most synchronization mechanisms can be divided into two broad categories: pes-

simistic and optimistic. Pessimistic synchronization makes the assumption that concurrent access to shared data is dangerous and should be prevented. Locks are used in such a way that only one thread at a time is allowed access to shared data[1]. This mechanism is called Mutual Exclusion because each critical section excludes other critical sections from executing concurrently.

Optimistic synchronization makes the assumption that, in most cases, concurrent threads accessing shared data will not conflict. Threads are allowed to proceed concurrently—both reads and writes are allowed to start at any time. If a conflict is detected one or more threads is delayed or rolled back and retried so the conflict can be avoided. The effect of the delays or rollbacks is the same as mutual exclusion: only one thread (or multiple read-only threads) is allowed to access the data at a time.

The effect of both pessimistic and optimistic synchronization is that accesses to shared data are serialized. Not only do the synchronization mechanisms add overhead and thus decrease performance, the serialization forms a sequential bottleneck that limits scalability.

Relativistic programming is a synchronization mechanism that does not fit neatly into either pessimistic or optimistic categories. Relativistic programming is optimistic in the sense that reads can start at any time without waiting for writes, and reads can run concurrently with writes to the same data. But unlike optimistic techniques there are no rollbacks nor retries—read operations always complete and hence are deterministic. Like pessimistic approaches, relativistic writers preserve data invariants so readers can assume the invariants hold. Unlike pessimistic techniques, relativistic readers are allowed to proceed concurrently with writers—even writers to the same data.

[1]As discussed in Chapter 2, some locking mechanisms partition data and one thread is allowed per partition. Other mechanisms allow multiple threads provided they are all reading but not modifying the data

Most synchronization methods create a total order on operations on a given data object. Locking methods create this order based on the order of lock acquisition. The locking primitives that bracket each operation have to guarantee the existence of this total order. For transactional memory, the commit process guarantees this order by serializing conflicting transactions.

Relativistic programming does not require a total order on operations on a given data object. Instead, readers are allowed to see writes as happening in different orders. Where necessary, writers specify which operations require a total order. Those operations for which the writer did not specify a total order are allowed to be seen in a different order by each reader. Because reads run concurrently with writes, readers can observe a different order of whole operations, they can also potentially observe a different order of steps within an operation (e.g. the individual stores to memory involved in adding a node to a tree). Because writers cannot exclude or delay readers, writers must guarantee that readers can only see correct ordering. Chapter 3 defines the criteria by which an order is considered correct. As a simple example, if a writer initializes a node and then publishes a pointer to that node, the writer must delay publishing the pointer until it is guaranteed that any properly formed reader that sees the pointer will also see the initialization.

Much work has already been done on the relativistic programming methodology. There are production quality implementations of relativistic programming primitives in both user mode [17, 16] and within the Linux kernel [46, 45]. Relativistic techniques are widely used within the Linux kernel for updating linked lists and other data structures. But there is much work yet to be done. In particular, relativistic techniques have primarily been applied to linked lists or other simple data structures that only require simple, single-node updates. Also, much of the work on relativistic programming has focused on read-side performance and scalability. Little attention

has been paid to scalability on the write-side other than partitioning techniques.

My thesis is that relativistic programming can be applied to complex data structures that require multi-node updates, and that relativistic reads can be combined with transactional writes to get the performance and scalability of relativistic reads and the automatic disjoint access parallelism of transactional writes. After presenting background information in Chapter 2, this thesis is developed as follows:

Chapter 3 explores the ordering requirements of relativistic programs by comparing them to the ordering requirements of reader-writer locking programs. This exploration leads to correctness criteria applicable to relativistic programs, provides a clear definition of what is required of the various relativistic primitives, and also provides foundational understanding allowing the further development of relativistic programming in the following chapters.

Chapters 4 and 5 apply the ordering principles defined in Chapter 3 to a Red-Black Tree (RBTree). RBTrees present a much more complex problem than previously developed relativistic data structures because RBTrees require multi-node transformations. These chapters show how the RP primitive originally intended to be used for safe memory reclamation can be used to order the visibility of updates such that the relativistic correctness criteria can be preserved throughout these multi-node updates.

Chapter 6 shows how the ordering requirements of relativistic programming can be maintained within a Software Transactional Memory (STM) system. Such an STM allows reads to proceed relativistically outside the STM system while updates are performed within the STM. This combination allows for high performance, scalable relativistic reads while still allowing for the automatic disjoint access parallelism of the STM on the write side. This chapter also provides a preliminary analysis of the viability of performing the lookup phase of a write transaction relativistically (outside the transactional system). The purpose of the relativistic lookup phase is

to both speed the lookup phase and reduce conflicts during the write phase of a transaction.

To summarize, this research contributes the following to the field of relativistic programming:

1. A new analysis of the ordering requirements of relativistic programs and the primitives that support them (Chapter 3)

2. Correctness criteria that can be applied to relativistic programs (Chapter 3)

3. A demonstration that relativistic programming can be applied to complex data structures including those requiring multi-node updates (Chapters 4-5)

4. A demonstration that the RP primitive originally intended for safe memory reclamation can also be used to order the visibility of updates (Chapter 4)

5. A mechanism by which high performance, scalable, relativistic reads can be combined with disjoint-access parallel writes using a modified software transactional memory system to perform the writes (Chapter 6)

Chapter 2

Background

This chapter begins with an explanation of four synchronization mechanisms: locking, Non Blocking Synchronization (NBS), Transactional Memory (TM), and Relativistic Programming (RP). The intent is not to give a complete exposition of these mechanisms, but rather to give an adequate presentation of the background so the particular features of RP described later can be put in the context of the other mechanisms. Section 2.2 describes some of the complexities of concurrent programming on modern hardware and Section 2.3 describes correctness criteria that are used with the different synchronization mechanisms. Section 2.3 also includes a proposal for new correctness criteria to be applied to relativistic programs.

2.1 Synchronization Mechanisms

2.1.1 Locking

Locking preserves data-safety by restricting concurrency. Regions of code that access shared data, called critical sections, are protected by locks in such a way that only one thread at a time can enter a critical section. Locking eliminates data races but at the expense of concurrency and therefore scalability. Some degree of scalability can be obtained through reader-writer locking [14, 50, 43] or fine-grained locking [8, 38, 60, 12]. Reader-writer locking allows multiple threads to proceed concurrently provided

they do not update the data. Fine grained locking partitions a data structure and assigns a separate lock to each partition. Multiple threads are allowed to access the data structure as long as no two are in the same partition.

As stated, locking preserves data-safety but at the expense of concurrency. Without concurrency, there can be no scalability because the amount of work done is limited to the amount of work that can be done by a single processor. Any time concurrency is limited, scalability is limited as well. Locking affects scalability because access to the data is serialized. It also affects performance because of the expense of acquiring a lock [4, 60].

Reader-writer locks were developed in an attempt to allow read-mostly workloads to scale by allowing read-side concurrency. However, most reader-writer lock implementations use atomic read-modify-write instructions to update a common lock variable. Contention for the lock variable and the serialization caused by atomic read-modify-write instructions limit both performance and scalability. Lev et. al. [43] show that the production Solaris reader-writer lock does not scale at all for certain read-only workloads. Triplett et. al. [62, 63] show that the production Linux reader-writer lock does not scale at all for certain read-only workloads. Chapter 3 presents benchmarks that document this effect for other reader-writer lock implementations.

There have been many attempts to improve performance and scalability of reader-writer locks. Hsieh and Weihl [37] proposed a reader-writer lock that used per-processor locks for readers. This approach avoids contention between readers allowing good scalability for read-only work loads. However, a writer must acquire all the read-locks. As the number of readers increases, the cost to a writer for acquiring all the read-locks becomes exorbitant. Read performance suffers significantly in the presence of writers because readers are excluded for the duration of the write, and writes are slow because writers have to acquire a potentially large number of read-locks.

Lev et. al. [43] present a reader-writer lock based on a Scalable Non-Zero Indicator (SNZI). Their observation is that writers do not need to know the number of readers, only if there are zero or not zero. They construct a tree of SNZI objects where the root of each subtree knows whether the subtree contains zero or non-zero readers. A node only needs to communicate with its parent when it moves in or out of its zero state. Readers can be distributed throughout the tree so the number of other readers they contend with is minimal. Writers can check the root of the tree for the existence of readers. Their implementation scales for read-only workloads but suffers significantly with even 1% writers.

Fine grained locking is another attempt to increase concurrency. If a data structure can be partitioned, a separate lock can be applied to each partition. Multiple threads can access the data concurrently provided each thread is accessing a disjoint set of partitions. Some data structures partition easily. For example, hash tables are often implemented with per bucket locking. Other data structures partition easily, but doing so has negative performance implications. The atomic read-modify-write instructions that are used in lock acquisition not only serialize access to the lock variable, they also are slow—as much as several orders of magnitude slower than non-atomic instructions [9]. If a linked list is partitioned such that each node has a separate lock, then the cost of acquiring a lock for each node can outweigh any gains through additional concurrency.

Locking has other, more subtle problems such as priority inversion [5, 39]. If a low priority task obtains a lock that is then required by a higher priority task, the higher priority task waits for the low priority task to complete and release the lock. A series of medium priority tasks that do not require the lock can prevent the low priority task from completing so the high priority task can have an unbounded delay. Solutions to this problem include boosting the priority of the task that holds the lock

whenever a higher priority task requests the lock. These solutions involve changes to the scheduler. Specifically, the scheduler needs to be aware of locks, and acquiring a lock may require checking scheduling information that adds to the overhead of lock acquisition. These changes to the scheduler and lock primitives increase the cost of lock acquisition causing a reduction in performance and scalability.

Finally, when using fine grained locking (or any scheme involving multiple locks), care must be taken to avoid deadlock [23, 22, 66] where multiple threads are waiting for each other to release locks. A common mechanism to avoid deadlock is to assign a total order on locks (for example, by using the address of the lock variable) and requiring all threads to acquire locks in the same order. Given the total required set of locks may not be known until after some of the locks are acquired, it may be difficult to define a total order on locks that all threads follow.

To summarize, locking has the advantage of being well understood. Its quirks are well documented in the literature, and there are many production systems that use locking successfully. Locking has the disadvantage of limiting both performance and scalability.

2.1.2 Non-Blocking Synchronization

With locking, a single thread that holds a lock can prevent progress in all other threads. A variety of techniques, collectively called Non-Blocking Synchronization[1] (NBS), were developed to address this shortcoming [31]. While liveness, the property where threads continue to make progress, was the original goal of these techniques, over time a greater emphasis has been placed on concurrent, scalable performance. To this end, a number of algorithms were developed for concurrent, non-blocking data

[1]There are a variety of techniques labeled in the literature Non-Blocking, Lock Free, and Wait Free. I am lumping all of these together under the name Non-Blocking. The specific differences between Non-Blocking, Lock Free, and Wait Free are not relevant to my discussion so it is convenient to refer to all these techniques by a single name.

structures [21, 10, 51, 24].

NBS algorithms tend to be extraordinarily complex for two reasons: They must accommodate any arbitrary interleaving of the memory operations from different threads; and they must preserve the liveness property in the presence of arbitrary delays in any other thread. The problem of interleavings can be thought of as follows: with locking, a programmer only needs to reason about correctness at the critical section boundaries. Without locking, the programmer needs to consider correctness at the boundary of every CPU or memory operation. The problem is made more complex by the out-of-order execution units of modern processors and the weak memory ordering properties they have [52, 53, 26, 27]. One cannot assume the program will execute in program order (Section 2.2 and Chapter 3 discuss this in greater detail). These considerations make it difficult to construct correct NBS algorithms.

The liveness property requires that no thread be able to put a data structure in a state where other threads will be blocked. If a particular update requires multiple steps (as opposed to a single atomic step), then the updating thread must leave a mark indicating the update is in progress. The mark must include enough information for another thread encountering the mark to either complete or abort and retry the operation. The marks typically have to be placed using atomic read-modify-write instructions, which means placing a mark has the same performance and scalability cost as acquiring a lock. The updating thread has to do the extra work to leave the marks. All threads, whether readers or writers, must check for marks and perform extra work when one is found. Finally, there is the potential for multiple threads to be working on the same operation—when multiple threads encounter the same mark, they all simultaneously attempt to complete or abort the operation. This "helping" slows the operation because multiple threads contend for the same memory addresses.

The abort and retry mechanism in NBS algorithms means NBS algorithms rarely

perform better than fine grained locking[2]. With fine grained locking, conflicting operations are prevented. With NBS, potentially conflicting operations are allowed to proceed optimistically, but only non-conflicting sets are allowed to complete. The others are rolled back and retried. So NBS offers deadlock avoidance and can provide disjoint access parallelism, but it rarely offers better performance or scalability than fine grained locking.

To summarize, NBS represents an attempt to increase concurrency over locking. At best it can offer the performance of fine grained locking. In reality, it suffers from complexity (it is difficult to develop correct algorithms) and performance problems.

2.1.3 Transactional Memory

Transactional Memory (TM) was developed to address some of the complexity issues of NBS [24, 25, 34]. NBS approaches tend to follow a pattern: before an operation (a read or write), the state of the data structure is saved; at the conclusion of the operation, the state is checked to validate that no conflicting updates were made during the course of the operation; the changes are either committed or rolled back depending on whether conflicts were detected. Transactional memory encapsulates this pattern rather than requiring an implementation unique to each data structure. The programmer simply starts a transaction, then performs a series of memory operations for which the TM system will save state, then requests that the TM system commit the changes. At this point, the TM system will check for conflicts and either commit or roll back the transaction. The commit is such that the entire transaction appears to happen atomically to other transactions—they will either see all the changes in the transaction or none of them.

[2]The exception to this rule is when the data object fits in a single word. Fine grained locking would contend for the lock variable and then update the data item. NBS could combine both of these operations for potentially better performance.

Conflict resolution compares the read and write sets of all current transactions. Different conflict detection mechanisms are possible, but in general, a conflict exists any time one transaction is reading a value that is being written by another, and one (or both) conflicting transactions will be delayed or rolled back and retried. Because conflict detection is based on the actual read and write sets of the transactions, TMs provide automatic disjoint access parallelism. That is, two transactions that access disjoint portions of a data structure are allowed to proceed in parallel because they do not conflict with each other[3].

TM solves the complexity problem inherent in most NBS solutions. Because transactions appear to execute atomically, the number of states a programmer needs to reason about is similar to a locking solution. However, TM suffers the same performance issues as NBS [13]. In addition, TM has difficulty handling non-idempotent operations (such as I/O) because they cannot simply be rolled back and redone when a transaction fails to commit [56, 47, 28].

At one time, it was hoped that transactional memory would be implemented in hardware [34, 32] to solve the performance problems. There have been some experimental hardware transactional memory systems, however none of these systems have made it to mainstream commercial availability. There are a wide variety of Software Transactional Memory (STM) solutions available [18, 19, 15, 20, 42]. There are so many systems because there are a variety of ways to attempt to solve the performance problems. Four parameters that are addressed in different ways by various systems are atomicity, time of update, meta-data, and visibility of reads.

Weakly atomic systems allow some accesses to be made non-transactionally. There is no isolation between the transactional and non-transactional accesses. The non-transactional accesses could cause a transaction to observe an inconsistent state. The

[3]Many transactional memory systems use hashing when computing conflicts. Hashing can result in false conflicts that would reduce the amount of disjoint access parallelism.

non-transactional accesses provide for higher performance but at the risk of violating the isolation guarantees of the transactional system. Strongly atomic systems guarantee that all accesses are within the transactional system. Typically, if an access is not explicitly wrapped in a transaction, an implicit transaction is created for each access. This increases overhead but provides stronger isolation guarantees.

Some systems update memory in place and keep an undo log which gets used to roll back transactions that fail. Updating in place makes commits inexpensive (the work of committing is already done), but it complicates precommitted threads because they observe some data that may eventually get rolled back. Other STM systems keep updates in a re-do log which gets used to update publicly visible memory when the transaction commits. The re-do log makes committing more expensive, but it means other threads will only see committed data.

Transactional memory systems keep meta-data on each object to facilitate isolation and atomicity. Some systems track individual memory locations (typically word sized memory locations, not byte sized), others track higher level objects. To reduce the amount of meta-data, many systems track a hash rather than the object itself. For example, a word based system could maintain meta-data on a block of words rather than separate data on each word. Hashing can improve performance because of the reduced cost of maintaining the meta-data. However, hashing can result in collisions—multiple objects hash to the same meta-data. These collisions are a form of false sharing that results in the transactional system reporting conflicts where none actually exist. These false conflicts reduce performance because of a higher incidence of failed transactions.

Some systems allow read-only transactions to proceed without creating any meta-data visible to other transactions. Invisible reads reduce the work involved with reading and should therefore result in better read performance. Invisible reads mean

a read-only transaction can never invalidate a writing transaction. Because read-only transactions do not create writer visible meta-data, they do not reserve or own the data they have accessed so writers can update the data during a read transaction. Because the data is not reserved, the read sets must be validated at commit time to ensure the read set represents a consistent snapshot of the data.

There are several mechanisms used to improve transaction performance. Some researchers have proposed mechanisms (called privatization) for temporarily pulling some data out of the transactional system so a single thread can more efficiently access the data without paying the transactional overhead [59, 44]. Others have proposed performing the actual data updates non-transactionally [33]. Each update has to provide an inverse function so the update can be undone if the transaction gets rolled back. The transactional system tracks operations rather than objects. Bronson et. al. [11] propose a mechanism for sets and maps where a mapping from a key to a single location can be computed non-transactionally. Given that mapping, set and map operations consist of a single read or write to that location. If that single read or write is performed transactionally, the individual transactions are small so the transactional overhead is small, but the operations still maintain the isolation, atomicity, and composability properties of the STM.

To summarize, TM encapsulates the complexity problem inherent in most NBS techniques. TM provides an easier mechanism to generate correct concurrent solutions including automatic fine grained conflict analysis. However, TM still suffers from poor performance and provides no more concurrency than is theoretically available with fine grained locking. Research is on-going on how to improve the performance of STM systems.

2.1.4 Relativistic Programming

The techniques now becoming known as relativistic programming were originally developed by Paul McKenney [46] and Jack Slingwine under the name Read-Copy-Update (RCU). The goals of RCU were to allow reads to proceed with low overhead and a minimum of synchronization. In particular, reads do not have to wait for writes even if the write is to the same data being read. Because of its performance characteristics, RCU is gaining widespread use in the Linux kernel. Figure 2.1 shows that the use of RCU within the Linux kernel is continuing to grow and has exceeded that of reader-writer locks. RCU is not limited to the Linux kernel: there is also a user mode library [17, 16] that is portable and is gaining use in non-kernel applications.

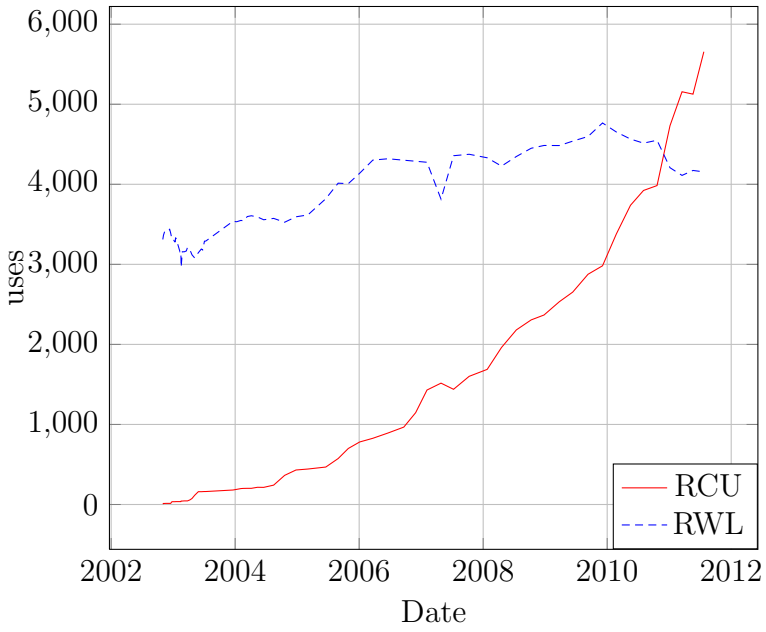


Figure 2.1: Count of the number of uses of RCU primitives and Reader Writer Locking primitives in the Linux Kernel.

McKenney described RCU as "a way of waiting for things to finish" [49]. This definition can be illustrated through the canonical use of RCU: When a writer removes a node from a linked list, a concurrent reader may still have a reference to the deleted

node. Before the writer reclaims the memory for the node, it waits for all current readers to finish. New readers cannot obtain a reference to the node (there is no link to it), so once all existing readers have finished, it is safe to reclaim the memory. By deferring reclamation, the writer ensures that concurrent readers will only see valid data.

Another mechanism RCU writers use to keep the data always-valid is copy-on-update. When a node needs to be changed, the writer makes a copy of the node in its private memory. The changes are made to the copy and once the changes are complete, the new version is swapped with the old by publishing a pointer to the new version. Copy-on-update prevents readers from seeing partially complete changes.

RCU shifts the cost of synchronization away from readers by approximating the delay required for safe reclamation (called a grace period in RCU literature). In order for memory reclamation to be safe, all readers that obtained a reference to the deleted node must drop that reference before the node is reclaimed. But RCU does not directly track references. Instead, RCU grace periods are defined such that any read-section that existed at the beginning of a grace period must finish before the grace period expires. With this definition, a grace period may be longer than necessary (some read-sections may not have a reference to the deleted node), but it is clearly long enough to make memory reclamation safe. See Chapter 3 for an expansion of this claim.

Some versions of RCU further approximate a grace period by not tracking read-sections, but by tracking alternate events. For example, Classic RCU does not allow read-sections to be preempted and prohibits read-sections from executing blocking calls. Because read-sections cannot be preempted or blocked, context switches cannot occur during a read-section, so context switches can be used to define grace periods. If every processor performs a context switch during a grace period, then every read-

section that existed at the beginning of the grace period must have finished. This approximation may wait for longer than necessary—multiple read-sections may occur between context switches—but it is guaranteed to wait at least long enough. For non-preemptible kernels, the primitives which bound read-sections add no overhead to readers. They can be completely optimized away because read-sections are not explicitly tracked.

Relativistic programming is a generalization of the techniques developed for RCU. If RCU can be described as "a way of waiting for things to finish", then relativistic programming can be described as "a way of ordering things." The relativistic programming primitives allow readers and writers to constrain the order in which the individual memory operations that compose the read and write are visible to each other. Unlike other synchronization mechanisms that try to impose a total order on operations to the same data, relativistic programming allows each reader to view writes in a different order. Each reader is allowed their own relative reference frame in which to view updates, and it is this property that gives relativistic programming its name.

Relativistic programming constrains ordering in a pairwise manner between a writer and each reader. Each reader forms a different pair with the writer so the ordering constraints can be applied differently to each reader-writer pair. By not requiring a total order agreed on by all threads, relativistic programming minimizes the ordering constraints and thus the overhead necessary to impose those ordering constraints. By minimizing the overhead, relativistic programming holds the promise for better performance and scalability.

Two examples of relativistic ordering will be given here. A complete treatment of relativistic ordering is given in Chapter 3. The first example involves deleting a node from a data structure. Any readers that observed the node must be prevented from

seeing the memory after it has been reclaimed. The relativistic approach does not constrain readers. Instead, the writer that deleted the node uses the `wait-for-readers` primitive between unlinking the node from the data structure and reclaiming the memory. The `wait-for-readers` guarantees that any thread that may have observed the node has finished before `wait-for-readers` returns. This use of `wait-for-readers` can be considered pair-wise because any reader that may have observed the node will affect the wait, but any thread that could not have observed the node because it started after the wait started will not affect the wait. Section 3.4.1 discusses a potential implementation of `wait-for-readers`.

The second example involves adding a node to a data structure. The writing thread creates and initializes the node and then links it into the data structure. Readers that observe the node must observe the initialization as happening before the node was made visible. Section 2.2 explains why program order cannot be relied on to preserve this ordering. However, if `rp-read` is used to read the pointer to the node and `rp-publish` is used to publish the pointer to the node, then any reader that sees the new node will also see the initialization. Notice that this is also a pair-wise ordering. The ordering of the initialization and publish is constrained for threads that observed the new node, but not for threads that did not observe the new node. Section 3.1.1 discusses how this is possible.

As relativistic programming has been generalized beyond RCU, it became advantageous to use different names for the primitives to make it clear that it is the general operation that is being referred to, not the specific RCU implementation. For example, the `synchronize_rcu` primitive in most RCU implementations waits for all preexisting readers system wide to complete their read-sections[4]. Sometimes this is done by waiting for a system event, such as a context switch on all processors, that

[4]The Linux SRCU implementation allows splitting SRCU into independent domains. Within Linux, the other implementations are system wide.

is guaranteed not to occur until after all readers have completed their read-sections. The RP primitive `wait-for-readers` must wait until any threads that hold references to particular data have released their references. The RCU implementations meet this requirement by meeting a broader requirement. Thus the RCU implementation is sufficient, but in general leads to longer delays than are strictly required for RP. The new primitive names are intended to make it clear that it is the concept of waiting for all readers who hold references to particular data that is meant, not a particular RCU implementation. A complete list of relativistic programming primitives is given below. Table 2.1 lists the RP primitives and their Linux kernel RCU equivalents.

`read-begin` This primitive marks the beginning of a read-section. In all known implementations, this operation is non-blocking and wait-free.

`read-end` This primitive marks the end of a read-section. In most implementations this operation is non-blocking and wait-free[5].

`rp-write-lock` This primitive marks the beginning of a write-section. Note that RP typically does not explicitly define how writers synchronize between themselves. It is often done with locking but this is not required. Within Linux, the `rcuupdate.h` header file contains a comment mentioning these primitives that explains that RCU readers do not synchronize with writers and that Linux leaves it up to developers to decide how to coordinate writers.

`rp-write-unlock` This primitive marks the end of a write-section.

[5]In the case of preemptible RCU in the Linux kernel, if the RCU read-section was preempted, then `read-end` will need to acquire locks to remove the task from the blocked-tasks list. But note that this is an implementation detail. `read-end` is not an inherently blocking action.

`rp-read` This primitive is used by readers when dereferencing a pointer. It guarantees that the pointer is fetched prior to being dereferenced (thus preventing value speculation) and it enforces dependent read consistency.

`rp-publish` This primitive is used by writers when updating a pointer. It guarantees that previous writes (in particular, those used to initialize the node being made public) are made visible to readers prior to the pointer being updated.

`wait-for-readers` This primitive is used by a writer when it needs to wait for all preexisting readers to finish. This is required between the time a node is removed from a data structure and the time when the memory for the node is reclaimed. The delay is required to ensure that no readers have a reference to the node.

`rp-free` This primitive asynchronously frees memory after all current read-sections have completed. By performing this operation asynchronously, the writer can complete the rest of its work before the grace period has expired.

Relativistic programming is distinct from the other synchronization mechanisms presented in this chapter in two ways:

1. Joint access parallelism is allowed between readers and writers.

2. By default, writes are not totally ordered with respect to all readers. Different readers can observe writes as happening in different orders.

RP name	RCU name	Reason for difference
`read-begin`	`rcu_read_lock`	This primitive does not acquire a lock and never blocks, so 'lock' should not be part of the name.
`read-end`	`rcu_read_unlock`	Same as above.
`rp-write-lock`	`rcu_write_lock`	Replaced the RCU prefix.
`rp-write-unlock`	`rcu_write_lock`	Same as above.
`rp-read`	`rcu_dereference`	Generalized the name from dereferencing a pointer to accessing any RP protected data.
`rp-publish`	`rcu_assign_pointer`	Generalized the name from assigning a pointer to publishing any form of RP protected data.
`wait-for-readers`	`synchronize_rcu`	Wanted to distinguish between the narrower "wait for readers who hold references to particular data" from the broader meaning implied by most current RCU implementations.
`defer-for-readers`	`call_rcu`	Used a name more descriptive of what the primitive does
`rp-free`	`kfree_rcu`	Short cut for `defer-for-readers(free)`

Table 2.1: Equivalence of relativistic programming and RCU primitives

The second item is what gives relativistic programming its name: each reader has a relative view of the order of events.

Much of the work on relativistic programming has focused on preserving read performance for the benefit of read-mostly data structures. Read performance is preserved by shifting the cost of synchronization primarily to the writers so readers can proceed almost as if they were sequential (single threaded) implementations. Individual write algorithms have been proposed for different data structures including linked lists and hash tables [62, 63]. More recently, a generalized construction technique has been proposed [61] that explains where to insert the `rp-publish` and `wait-for-readers` barriers into an RP algorithm in order to preserve the ordering requirements of relativistic programs. Much of this work has presumed the ordering properties of `wait-for-readers` and `rp-publish`. Chapter 3 presents an analysis of the ordering requirements of relativistic programs. This analysis lays the foundation for the research presented in later chapters.

2.1.5 Comparing the Approaches

Table 2.2 lists various synchronization mechanisms and the properties they supply. The read-read column indicates that multiple threads, all reading the same data, can proceed in parallel. The read-write column indicates that threads reading and writing the same data can proceed in parallel. The applicability column indicates how broadly the mechanism has been applied to data structures. The disjoint access parallelism column indicates whether the mechanism allows threads accessing disjoint portions of the data structure to proceed in parallel.

Fine grained locking allows disjoint access parallelism, but it is up to the programmer to determine how to partition the data. Some data structures are difficult to partition. NBS typically does not explicitly partition data, but can provide some

Synchronization Method	read-read	read-write	applicability	disjoint access parallelism
Relativistic Programming	yes	yes	limited	not specified[b]
Global Lock	no	no	broad	no
Reader-writer lock	yes	no	broad	no[a]
Fine grained locking	no[a]	no	limited	manual
Non-Blocking Sync	yes	no	limited	manual[c]
Transactional Memory	yes	no	broad[d]	automatic[e]

[a] Fine grained locking can be combined with reader-writer locking to provide both read-read concurrency and disjoint access parallelism.

[b] Synchronization between writers is not explicitly specified by RP, but various locking schemes are typically used including fine grained locking that allow manual disjoint access parallelism.

[c] NBS programmers need to manually detect and manage access conflicts. Most NBS algorithms do this in a way that accomplishes disjoint access parallelism.

[d] TM systems have trouble handling non-idempotent operations, but are in most other ways broadly applicable.

[e] Most high performance TM systems perform conflict detection on hashed values. Hashing collisions lead to false sharing which limits the amount of disjoint access parallelism obtainable by the TM system.

Table 2.2: Synchronization mechanisms and their properties.

level of disjoint access parallelism. However, the NBS programmer needs to explicitly manage all the complexity of concurrent accesses. TM automatically provides disjoint access parallelism. With relativistic programming, synchronization between writers is not explicitly specified. Various mechanisms are used including fine grained locking that provides manual disjoint access parallelism.

Fine grained locking has limited applicability because of the difficulty of partitioning some data structures. NBS has limited applicability because each data structure requires a separate and complex implementation. Relativistic programming has had limited applicability because it has mostly been applied to linked lists (or structures such as hash tables that are built on top of linked lists) and because each new data structure requires a new algorithm.

This dissertation addresses the two limitations of relativistic programming listed in Table 2.2. It shows that relativistic programming can be applied to a broad range of

data structures including those that require complex multi-node updates (see Chapter 4). It also shows how relativistic programming can be used in conjunction with software transactional memory to provide automatic disjoint access parallelism on the write side while supporting concurrent relativistic reading of the same data (see Chapter 6).

2.2 Impact of Reordering on Concurrent Programming

Programmers tend to assume that their programs execute in program order, but both optimizing compilers and modern hardware invalidate this assumption. Optimizing compilers transform source code into efficient machine code whose execution is equivalent to a sequential execution of the source code. But the key here is that the compiler assumes a *sequential* execution. Both the compiler and the underlying hardware can reorder code in a way that is safe for a sequential execution but may not be safe for a concurrent execution [9].

Modern hardware does not present a sequentially consistent interface to bare metal programmers. Multiple levels of cache, instruction pipelines, and out of order execution units allow modern CPUs to be fast, but these features mean concurrent programmers cannot assume sequentially consistent behavior of their code [2, 52, 53, 3].

Listing 2.1 shows code for two threads that run concurrently. Thread 1 initializes a node and Thread 2 spins waiting for the node to be initialized before examining its contents. For a sequential execution, it does not matter what order the fields are initialized. But for a concurrent execution, it is important that `init_complete` be initialized after the other fields. If the compiler assumes a sequential execution, it may reorder these lines, for example, to make better use of registers. The CPU may reorder the lines, for example, due to cache line availability. Finally, even if the compiler and CPU leave the order intact, the memory systems on some hardware

architectures can also reorder when the stores are written to memory.

thread1

```
1 void init(node_type *data) {
2    data->field1 = FIELD1_INIT;
3    data->field2 = FIELD2_INIT;
4    data->init_complete = TRUE;
5 }
```

thread2

```
1 while (!data->init_complete)
2 {}
3 examine(data);
```

Listing 2.1: The effects of Line 4 of Thread 1 must become visible after the effects of lines 2 and 3 to prevent Thread 2 from seeing uninitialized data.

The impact of reordering can be localized in lock based code. Use of locking differentiates between three categories of code: that accessing only thread-private data, that appearing in critical sections, and that comprising the locking primitives themselves. Local code can be treated as sequential because there is no shared data. The critical section can also be treated as sequential because mutual exclusion prevents conflicts involving writes. So only the locking primitives are truly concurrent.

A programmer writing new locking primitives must be concerned about all the complexities of concurrent programming. They must ensure that the primitives enforce the semantics of the lock on the target hardware. They must account for whatever reorderings the CPU and memory system allow because the lock primitive code is truly concurrent. They must also ensure that the locking primitives contain whatever barriers are necessary to prevent the compiler, CPU, or memory system from moving code across a critical section boundary; otherwise, code that should have been within a critical section can take effect outside that critical section.

If locking primitives work correctly and are used correctly, a programmer using those primitives in a multithreaded program can view both their local code and critical

section code as if it was sequential.[6] They still have many other issues to worry about, but not compiler, execution unit, nor memory system reorderings.

In contrast with locking programs, NBS, TM, and relativistic programs allow reads to run concurrent with writes. They are truly concurrent in the sense that they contain data races[7]. Because NBS, TM, and RP programs are truly concurrent, programmers need to be aware of and manage the reorderings discussed earlier. Each of these mechanisms handles this concurrency differently.

With NBS, there is no standard way to handle the effects of concurrency. The programmer has to deal directly with all the complexity of potential reorderings. It is for this reason that NBS algorithms tend to be very complex.

With TM, the complexity of concurrency is largely hidden, much as it is with locking. The TM system guarantees that all of the operations within a transaction will appear to occur atomically to any other transaction. As a result, TM programmers only need to reason about correctness at transaction boundaries much as locking programmers only need to reason about correctness at critical section boundaries.

Relativistic programming requires programmers to be directly aware of concurrency similar to NBS. However, relativistic programming identifies specific operations that need to be ordered and provides a small number of ordering primitives with simple rules for their placement. If these primitives work correctly and are used correctly, they will only allow correct orderings of execution. The orderings and primitive placement are the subject of Chapter 3.

Table 2.3 lists the four synchronization approaches and how visible the effects of

[6]Boehm [9] points out that in the absence of a memory model, a compiler may transform code so that data races are created that did not exist in the source. When this happens, code that the programmer thought was data race free (and therefore viewable as sequential) will in fact be concurrent. The keepers of the Java and C++ languages have addressed these issues by developing formal memory models for these languages.

[7]The broadest definition of a data race is two threads that access the same data concurrently where one of the accesses is a write. Some authors [6] restrict the definition to only accesses that are not protected by memory ordering primitives. The term is used here in the broader sense.

concurrency are. For locking, programmers define critical sections and wrap those sections in locking primitives. All the effects of concurrency are hidden by these primitives. TM is similar, but transactions are defined rather than critical sections. With NBS, all the effects of concurrency are visible to the programmer and the programmer must explicitly manage them. With RP, the effects are visible but the primitives and the methodology abstract away the details making it much easier to manage concurrency.

Synchronization Method	**Effects of Concurrency**
Locking	Hidden
NBS	Visible
TM	Hidden
RP	Visible but abstracted

Table 2.3: Visibility of the effects of concurrency

2.3 Correctness Criteria

Just as programming in a concurrent environment is more complex than in a sequential environment, so too is defining correctness. Concurrent correctness criteria must not only define the correct sequential behavior, they must also define correct interaction between threads.

Correctness of implementations that use locking is often defined in terms of data structure invariants. The invariants define a correct state of the data structure and can define properties of both the ADT and the underlying data structure used to represent the ADT. Some example invariants are given in Table 2.4. Threads are allowed to violate the invariants while they hold the lock, but the invariants must be restored prior to releasing the lock. Proving correctness consists of proving that all invariants are restored prior to releasing the lock. Mutual exclusion means that while one thread holds the lock, no other thread will interfere with its access of the data

and thus with its ability to restore the invariants. Mutual exclusion also means that while one thread has temporarily violated the invariants during an update, no other thread will access the data and see an invalid state.

Data Structure	**Invariants**
Linked List	Every node is reachable and connected to the root
Doubly Linked List	for each node: `node->prev->next == node`
	for each node: `node->next->prev == node`
Hash Table	Each element is reachable from the correct bucket
	Each element is reachable only from the correct bucket
Sorted Tree	All nodes on the left branch of a subtree have a key less than the the key of the root of the subtree
	All nodes on the right branch of a subtree have a key greater than or equal to the key of the root of the subtree
Map	Each key value occurs only once

Table 2.4: Potential invariants for various data structures.

NBS does not use critical sections nor mutual exclusion, so invariants cannot be used in the same way they are for locking. Data can be changed by one thread during another thread's operation so threads cannot even assume the constancy of data. The most common correctness criteria for NBS solutions is linearizability. Linearizability requires an equivalence between the linearizable computation and a legal sequential computation [36]. Linearizability is most often proven by showing that each operation has an atomic step at which the operation takes effect. Prior to that atomic step, the ADT is in the state prior to the operation; after that atomic step, the ADT is in the state following the operation. Because the step is atomic, threads either see the state before or after the operation.

Linearizability also allows reasoning about NBS solutions in terms of pre and post conditions much as locking allows reasoning about invariants [65]. Reasoning about pre and post conditions is easier than reasoning about every intermediate state in an NBS solution, and this is the value of linearizability. However, the easier reasoning comes at a cost. Herlihy et. al. [35] proved there is a performance cost to linearizability[8]. Their proof was not for a particular implementation, but covered any implementation.

Similar to lock based systems, transactional memory allows correctness conditions based on invariants. The programmer must prove that a transaction restores all the invariants prior to committing. The programmer then depends on the TM system to provide isolation and atomicity. Isolation guarantees that other transactions will not interfere with the current one. Atomicity guarantees that other transactions will observe either all of the effects of the current one or none of the effects of the current one. If the TM system provides sufficiently strong isolation and atomicity guarantees, then reasoning about a TM system is the same as reasoning about a lock based system.

Relativistic programming has not had adequate correctness criteria. The use of invariants enforced at the end of a write-side critical section is inadequate because there is no isolation between reads and writes. Readers can see the intermediate states including any that might temporarily violate the invariants. Linearizability is also inadequate because relativistic programming explicitly allows non-linearizable solutions. The name "relativistic programming" comes from the fact that each reader is allowed to have their own view of the order of updates. This relative view of updates is specifically disallowed by linearizability.

The following correctness criteria can be applied to relativistic implementations

[8]Their work dealt specifically with counting networks, but the underlying cause of the performance cost applies to other data structures

of Abstract Data Types (ADTs):

1. Updates leave the ADT in an always-valid state meaning a read can access the data structure at any time without the need for synchronization.

2. Read operations on the ADT see the effects of all previous non-concurrent updates.

3. Read operations do not see any of the effects of later non-concurrent updates.

4. For a read that is concurrent with an update, the read sees either the state of the ADT prior to the update or after the update, but the read is not allowed to see any other state.

Item 1 differs from reader-writer locking in that reader-writer locking writers can violate the data structure invariants while holding the write-lock. RP writers have the data always-valid because readers can access the data at any time. Note that it is the ADT that must be always-valid. Invariants of the underlying data structure may be temporarily violated provided the violation does not result in an invalid ADT state. For example, the balance properties of a tree may be temporarily violated provided that doing so does not lead to incorrect read operations.

Items 2–3 are the same as for reader-writer locking. They state that non-concurrent operations form a total order that matches the serial order.

Item 4 deals with read-write concurrency, and states that a write cannot affect an unrelated read. For example: if a set includes the values 1..5 then a write inserting the value 6 should not cause a concurrent read looking for one of the values 1..5 to return `not-found`. Item 4 is intended to allow non-linearizable behavior in that not all concurrent readers have to observe the same states. This aspect of relativistic programming is covered in Section 3.3. Also note that Item 4 refers to states of the

ADT, not states of the data structure implementing that ADT. For example, if a red-black tree is used to implement a Map, then readers may see intermediates states of the red-black tree, but each reader visible state of the red-black tree must map to the state of the Map either before or after the update.

Chapter 3

The Ordering Requirements of Relativistic Programs

As discussed in Section 2.2, both compilers and hardware are capable of reordering the execution of a program so it is not safe to assume the program executes in program order. Reordering is not a problem for sequential programs because compilers and hardware work together to preserve the effect of a sequential execution of the source code. Concurrent programs may observe reorderings in such a way that erroneous results are produced. For example, if a thread is adding a node to a structure, the source code might initialize the fields of the node and then set a pointer so the node is part of the structure. But the pointer assignment might get reordered with some portion of the initialization in such a way that another thread might encounter uninitialized data. Concurrent programs must be written in such a way that reorderings that lead to erroneous results are prevented.

This chapter examines the mechanisms used by reader-writer locking (RWL) and relativistic programming (RP) to control ordering. This chapter is outlined as follows: Section 3.1 introduces the terminology used to describe ordering relationships and also discusses the primitives used by RWL and by RP. Section 3.2 looks at four progressively more complex examples to illustrate how the RWL and RP primitives are used and to demonstrate how they control ordering. Section 3.3 looks at additional considerations that exist when considering multiple writers rather than a single writer.

Section 3.4 shows both analytically and empirically that the performance of RP is better than that of RWL. Section 3.5 summarizes the contributions presented in this chapter.

3.1 Introduction to Ordering Control

This section introduces the terminology used in this chapter to describe ordering relationships in concurrent programs. This section also describes the primitives used in RWL and RP programs to control ordering.

3.1.1 Terminology

Three types of ordering relationships are discussed in this chapter: *program order*, *occurred-before*, and *required-before*. *Program order* is the order defined by the source code of the program. As noted earlier, both compilers and hardware can reorder the execution of the code. Where reordering can cause erroneous executions, primitives must be inserted to preserve *program order*.

In a race condition where one thread writes a value that another thread reads, the read and write can occur in one of two orders[1]. If the read observes the value written, then we say the write *occurred-before* the read. If the read observes the value that existed prior to the write, then we say the read *occurred-before* the write. Note that there is no correctness implication with either observation. The *occurred-before* relationship simply documents the outcome of a race for two particular instances of threads. Other threads are allowed to make different observations. In the text, an arrow will be used to mean *occurred-before* so $A \rightarrow B$ reads A *occurred-before* B.

Occurred-before relationships often impose correctness constraints on other opera-

[1]This discussion assumes the values are read and written atomically. This is the case, for example, with word sized, word aligned values.

tions. Consider the case where a writing thread creates and initializes a node and then links that node into a data structure. A reading thread is traversing the data structure and reads the pointer written by the writing thread. If the write *occurred-before* the read, then the reader will observe the new node, and so it is important that the initialization off the node occur before the reader examines the node. We call this a *required-before* relationship and say that the initialization is *required-before* the dereference of the node. In general, program order cannot be depended upon to enforce *required-before* relationships; the existence of the *required-before* relationship implies that the progrmmer must do extra work to enforce the relationship. Note that the *required-before* relationship in this example only applies to threads for which the write *occurred-before* the read. Threads that do not have this *occurred-before* relationship also do not have the *required-before* relationship. In the text, a double arrow will be used to mean *required-before* so $A \Rightarrow B$ reads A is *required-before* B.

Correctness conditions often have the form: If A *occurred-before* B then C is *required-before* D. Both *occurred-before* and *required-before* relationships involve observations, so they are in this respect the same. The difference, as implied by the form of the correctness conditions, is that the *required-before* relationship implies the programmer must do extra work to guarantee this relationship is observed every time the *occurred-before* relationship is observed.

Note about notation: When discussing ordering relationships, multiple threads may be discussed in the same context. If the context does not make it clear which thread is meant, a subscript will be used on an operation name to specify the thread so $oper_{Thread\ 2}$ means the *oper* of Thread 2. Because long subscripts make the text awkward, the subscript may be abbreviated so *oper* of Thread 2 may be written $oper_{T2}$.

Ordering relationships can be depicted graphically as shown in Figure 3.1. Dashed

arrows indicate *program order* relationships. As in the text, single and double arrows indicate *occurred-before* and *required-before* relationships respectively. Figure 3.1 shows the relationships for readers accessing a list with adjacent nodes A and C and a writer inserting a node B between A and C. The write takes place in two steps: *init* initializes B and *link* links B into the list by changing the pointer out of A. There are two steps of a read that are important when considering the correctness of an implementation: in *obtain ref*, the reader obtains a reference to the node following A. This could be either a reference to B or C depending on whether or not *link* $\rightarrow$ *obtain ref*. The second step of the read is *deref* where the node whose reference was obtained in *obtain ref* is read. If *obtain ref* $\rightarrow$ *link* then the reader will not see B and there is no constraint on the *init* and *deref*. However, if *link* $\rightarrow$ *obtain-ref*, then the reader sees B and the relationship *init* $\Rightarrow$ *deref* must be enforced or the reader may see uninitialized data.

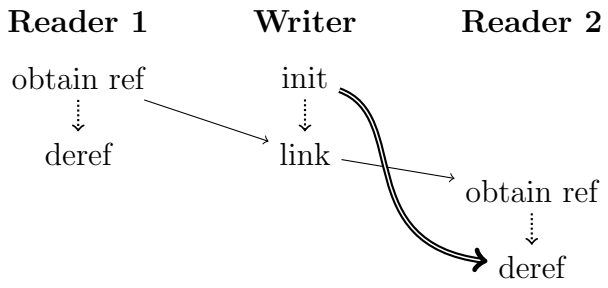


Figure 3.1: Orderings that have to be enforced by synchronization. Reader 2 obtains a reference to the new node, so the initialization of that node must be visible to Reader 2 before Reader 2 accesses that node.

Listing 3.1 shows two mechanisms for enforcing *required-before* relationships. In the first mechanism, Thread2 is waiting for an action in Thread1. The wait in combination with `barrier2()` enforces $A \Rightarrow B$. The barrier is required to prevent the compiler or the hardware from executing B prior to the `while`. The mechanism used by Thread2 will be referred to as the *wait* mechanism of enforcing *required-before*

relationships. In the second mechanism, Thread3 observes the second write from Thread1. The `barrier()` in Thread1 requires that the first write becomes visible before the second. The `barrier()` in Thread3 prevents the compiler or hardware from speculatively executing D prior to the `if`. The combination of these barriers enforces $A \Rightarrow D$ [57, 54]. This mechanism will be referred to as the *dependency ordering* mechanism of enforcing *required-before* relationships.

```
Thread1()           Thread2()           Thread3()
A: a=1;             while (!a)          if (c)
   barrier1();      {}                  {
C: c=1;             barrier2();           barrier3();
                    B:                  D:
                                        }
```

Listing 3.1: Thread2 uses a wait to guarantee $A \Rightarrow B$. Thread3 relies on the memory barrier in Thread1 to guarantee that $A \Rightarrow D$.

Listing 3.2 shows a special case of *dependency ordering*. The operation at D is a dependent read because it dereferences a pointer read earlier in the code. All modern hardware enforces dependent read consistency without the need for a hardware barrier. As a result, in this case, `barrier2()` requires a compiler barrier, but not a hardware barrier.

The *dependency ordering* mechanism waits on the memory system. The `barrier()` in Thread1 of Listing 3.1 can be thought of as waiting for the memory system to complete the work required by the first write in Thread1. The *wait* mechanism first waits for program logic. The `while` loop in Thread2 of Listing 3.1 is waiting for the program logic in Thread1 to write a value. After Thread1 writes the value, Thread2 must also wait for the memory system because it is the memory system that communicates the value to Thread2. Because the *dependency ordering* mechanism

```
Thread1()              Thread2()
A: a=1;                p = c;
   barrier1();         if (p != NULL)
C: c=&node;            {
                         barrier2();
                       D: d = p->value;
                       }
```

Listing 3.2: Thread2 implements to *dependency ordering* mechanism for enforcing *required before* relationships. Because the operation in D is a dependent read, `barrier2()` does not require a hardware barrier on any modern hardware.

waits only for the memory system but the *wait* mechanism must wait for both program logic and the memory system, the *wait* mechanism is a heavier weight mechanism. Its use will likely have a larger negative effect on both performance and scalability.

Reader-writer locking only gives access to the *wait* mechanism for enforcing ordering. Relativistic programming gives access to both mechanisms. Access to the lighter weight *dependency ordering* mechanism is part of what gives relativistic programming its performance advantage over reader-writer locking.

3.1.2 RWL Ordering Primitives

Reader-writer locking uses the primitives `read-lock`, `read-unlock`, `write-lock`, and `write-unlock` to control ordering. The semantics of reader-writer locks allow read-sections to proceed concurrently with other read-sections, but do not allow write-sections to proceed concurrently with any other sections (read or write). Stated using the ordering relationships:

1. if *read-lock* $\rightarrow$ *write-lock* then *read-unlock* $\Rightarrow$ *write-lock*
2. if *write-lock* $\rightarrow$ *read-lock* then *write-unlock* $\Rightarrow$ *read-lock*
3. if *write-lock*$_i$ $\rightarrow$ *write-lock*$_j$ then *write-unlock*$_i$ $\Rightarrow$ *write-lock*$_j$

The first constraint prevents a writer from starting if any readers hold the lock. The second constraint prevents a reader from starting if a writer holds the lock. The third constraint prevents a second writer from starting if another writer holds the lock. Note that there is no constraint to prevent a second reader from starting if another reader holds the lock. The absence of this constraint is what allows read-read concurrency.

If a thread holds the write-lock, another thread that calls either *read-lock* or *write-lock* must wait for the write-lock to be released. For this reason, both the *read-lock* and *write-lock* must use the *wait* mechanism of enforcing *required-before* relationships.

3.1.3 RP Ordering Primitives

RP has five ordering primitives: `rp-publish`, `rp-read`, `start-read`, `end-read`, and `wait-for-readers`. The `rp-publish` and `rp-read` primitives work together to implement the *dependency ordering* mechanism. The writing thread uses `rp-publish` for the second of the two writes, and the reading thread uses `rp-read` to read the published value. The `rp-publish` contains compiler and hardware barriers to prevent reordering across the `rp-publish`. RP assumes that the read following `rp-read` will depend on the value acquired through the `rp-read`. Because all modern systems enforce dependent read consistency without requiring a memory barrier, the `rp-read` forces the compiler to emit a load instruction, but it has no other effect on ordering. With RP, most of the work required to guarantee ordering is placed on writers. For this reason, `rp-read` will not be mentioned in most of the discussion on ordering, however, Section 3.2.5 mentions the rule for when to apply `rp-read` and this rule applies to all platforms, whether they include a memory barrier in `rp-read` or not.

The `wait-for-readers` primitive guarantees that any reads that started prior to

the wait will finish before the end of the wait. The `start-read` and `end-read` are used to bound read-sections to facilitate `wait-for-readers`[2]. Stated using the ordering relationships (see also [17]):

$\forall$ readers and any `wait-for-readers` if `start-read` $\rightarrow$ (the start of `wait-for-readers`) then `end-read` $\Rightarrow$ (the end of `wait-for-readers`)

This definition makes a distinction between the start of `wait-for-readers` and the end of `wait-for-readers` because `wait-for-readers` has a duration. Some portion of a read-section may be concurrent with the `wait-for-readers`, but any read-section concurrent with the beginning of the `wait-for-readers` must be complete before `wait-for-readers` terminates. Note that the definition of `wait-for-readers` does not comment on reads that begin after the start of the wait. This is intentional because `wait-for-readers` does not need to wait for these reads.

Because `wait-for-readers` implements the *wait* mechanism, it must include whatever compiler and hardware barriers are required to prevent reordering across it.

3.2 Case Studies

Four operations will be used to illustrate ordering relationships. A singly linked list will be used to illustrate these operations here, but the operations are representative of operations on a wide variety of data structures. Chapter 4, for example, shows that all the operations required to maintain a red-black tree can be described in terms of these four basic operations. The operations are insert, delete, move forward, and move back. The insert and delete operations are self explanatory. The move forward and move back operations consist of moving a node from one location within a data

[2]Some RP implementations do not use `start-read` and `end-read` to compute the `wait-for-readers`'s delay [17]. However, such implementations still support `start-read` and `end-read` for software-engineering reasons.

structure to another. The direction is relative to the traversal order of readers.

3.2.1 Insert

There are two steps in adding a node to a linked list. The node must be initialized and the node must be linked into the list. Consider a linked list with two nodes labeled A and C. A write thread adds node B between nodes A and C. The following names are used for specific steps in the insert whose order is critical:

obtain ref a reader obtains a reference to the node following A. This could be a reference to B or C.

deref the reader accesses the contents of the node whose reference was obtained by *obtain ref*. This includes dereferencing the pointer to the next node.

init the writer initializes node B.

link the writer links B into the list by changing the pointer out of A.

The ordering relationship required for correctness is given by:

if *link* $\rightarrow$ *obtain ref* then *init* $\Rightarrow$ *deref*

Figure 3.2 shows how reader-writer locking primitives maintain the correct ordering relationships for an Insert operation. The inter-thread relationships are defined by reader-writer locking semantics. For Reader 1, *read-lock* $\rightarrow$ *write-lock*. This imposes the relationship *read-unlock* $\Rightarrow$ *write-lock*. For Reader 2, the relationship *write-lock* $\rightarrow$ *read-lock* requires *write-unlock* $\Rightarrow$ *read-lock*.

The correctness of a reader-writer lock implementation depends on program order dependencies being preserved. In particular, because *init* and *link* do not have any

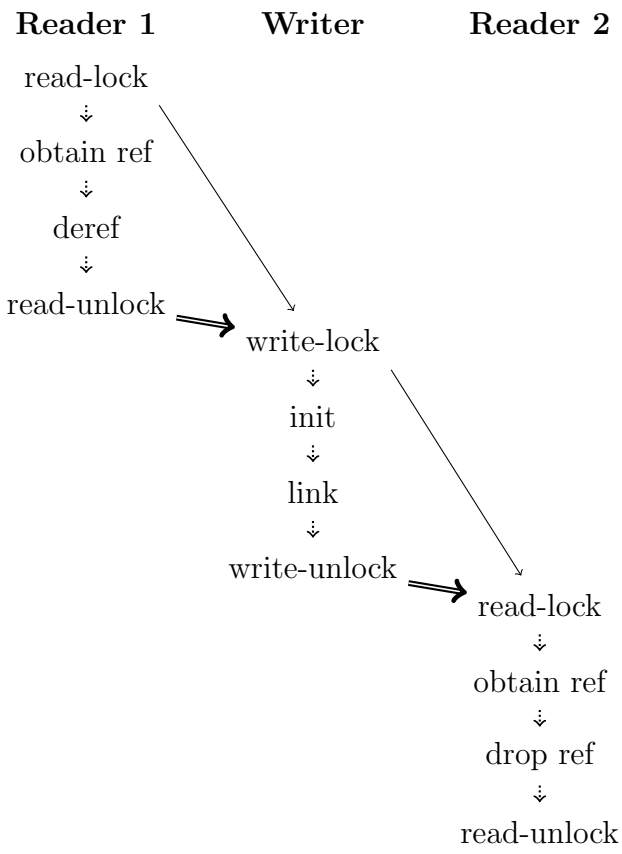


Figure 3.2: Ordering relationships for a reader-writer lock insert operation. The locking semantics prevent the write from being concurrent with a read. The lack of concurrency is what enforces the ordering relationships.

required-before relationships specified, the compiler or hardware can re-order them. If, in addition, *init* was delayed until after *write-unlock*, then Reader 2 could see uninitialized data. Other examples could be given to show the need to prevent re-ordering around the other primitives, so the locking primitives must not only enforce the locking semantics, they must also contain whatever compiler and memory barriers are required to guarantee the contents of the critical section do not leak outside. If the locking primitives preserve these properties, then only correct orderings will be allowed.

Figure 3.3 shows the ordering relationships for a relativistic insert. Because both *init* and *link* are memory writes and Reader2 observes the second of the two, the relationship $init \Rightarrow deref_{R2}$ can be satisfied using the *dependency ordering* mechanism. The `rp-publish` in Line 5 of Listing 3.3 guarantees that the memory writes that are part of the initialization are visible to readers prior to the new node being reachable by readers. In other words, using `rp-publish` to perform the *link* and using `rp-read` to *obtain ref* will guarantee $init \Rightarrow deref_{R2}$.

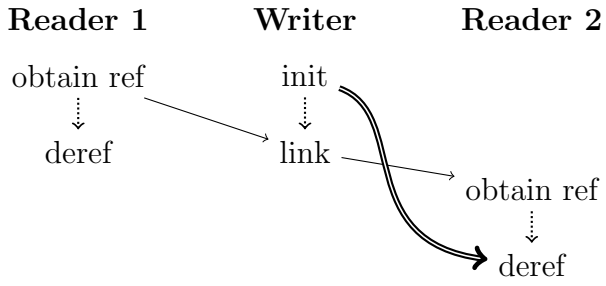


Figure 3.3: Ordering relationships for a relativistic insert.

```
1   void insert_at(prev, key, value)
2   {
3      node = create_and_init(key, value);
4      node->next = prev->next;
5      rp_publish(prev->next, node);
6   }
```

Listing 3.3: Pseudo-code for a relativistic insert.

Figure 3.3 is the same as Figure 3.1, which shows the ordering relationships independent of synchronization mechanism. The similarity of these two figures shows that, at least for the insert operation, RP imposes only the minimum of ordering constraints.

Note the following in comparing the RWL and RP insert operations: First of all, the net effect of both implementations is the same: a given reader sees the list either

before or after the insert. In neither case are any intermediate states visible. Secondly, the graphical representation of the operations in Figures 3.2 and 3.3 gives a hint as to their comparative performance. The RWL implementation takes up more space indicating it will likely be slower. Also, the RP implementation allows concurrency. The readers overlap with the writer. Finally, for Insert, the `start-read` and `end-read` primitives are not part of the ordering requirements. Their use will be demonstrated in the next section on the Delete operation.

3.2.2 Delete

This section examines a linked list delete operation. The linked list begins with three nodes labeled A, B, and C. A write thread removes node B leaving nodes A and C and then reclaims the memory used by B. The following names are used for specific events whose order is critical:

obtain ref Obtain a reference to the node following A. This could either be a reference to B or C depending on whether the reference is obtained before or after B is removed from the list.

drop ref The thread no longer has a reference to the node following A. In most cases, code does not explicitly drop references, so *drop ref* is implied after the last use of the reference in question.

unlink The writer removes B by changing A so it points to C instead of B.

reclaim The writer reclaims the memory used by B, allowing this memory to be used for another purpose.

The ordering relationship required for correctness is given by:

if *obtain ref* $\rightarrow$ *unlink* then *drop ref* $\Rightarrow$ *reclaim*

The analysis of the reader-writer implementation of Delete is the same as for Insert. There is no concurrency between readers and writers so if *obtain ref* $\rightarrow$ *unlink*, then the entire reader occurs before the writer thus guaranteeing *drop ref* $\Rightarrow$ *reclaim*. The only requirement is that the `lock` and `unlock` primitives include compiler and memory barriers to prevent the contents of the critical section from leaking outside.

Figure 3.4 shows the ordering for a relativistic Delete. The `wait-for-readers` is depicted in two parts—the beginning of the wait and the end of the wait. Splitting the wait makes it easier to identify the occurred-before and required-before relationships in relativistic algorithms.

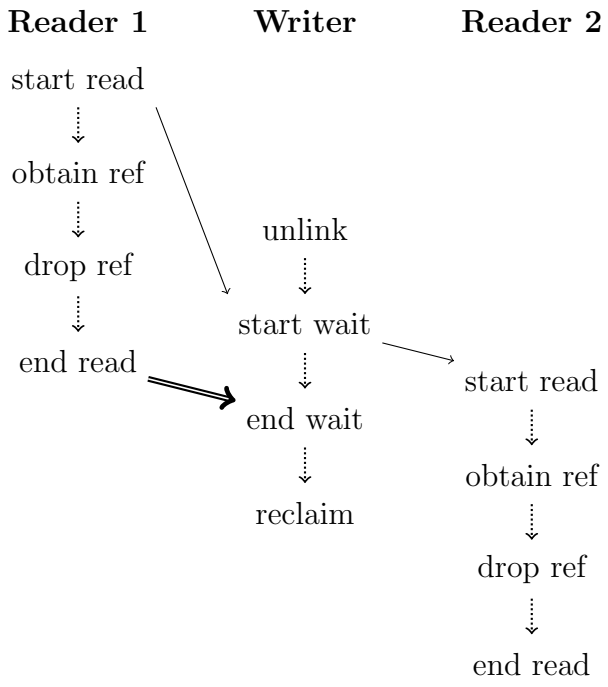


Figure 3.4: Ordering relationships for a relativistic delete. The `wait-for-readers` was split into *start wait* and *end wait* to make the relationships more clear.

There is no explicit ordering relationship between *obtain* ref_{R1} and *unlink*. They can occur in either order. This can be visualized in Figure 3.4 because the *unlink* can be raised above *obtain* ref_{R1} without affecting any other ordering relationships. How-

ever, any reader that has *obtain ref* → *unlink* must also have *start read* → *start wait*. The converse is not true: Having *start read* → *start wait* does not imply any relation between *obtain ref* and *unlink*. As a result, using *start read* → *start wait* to limit ordering is more conservative than using *obtain ref* → *unlink* because `wait-for-readers` may wait longer than necessary, but it is guaranteed to wait long enough. That is, it is guaranteed to wait for all threads that have *obtain ref* → *unlink*, but it might also wait for some threads that have *unlink* → *obtain ref*. But because it is guaranteed to wait for all threads that have *obtain ref* → *unlink*, it enforces the correctness criteria *drop ref* ⇒ *reclaim* on all threads to which this is required.

In order for the above argument to hold, the *start read* and *end read* primitives require compiler and memory barriers to prevent *obtain ref* from occurring before *start read* and to prevent *drop ref* from occurring after *end read*. These are the same barrier requirements for reader-writer locking primitives. Similarly, the `wait-for-readers` primitive must have compiler and memory barriers to prevent *reclaim* from occurring before *end wait*. With these barriers in place, the correctness condition is met.

Note the following in comparing the RWL and RP implementations. First, much like with Insert, in both cases the reader observes either the state before or after the Delete, but no intermediate states. Second, the RP implementation used the *wait* mechanism to enforce the ordering relationships much as the RWL implementation. However, the RP implementation still allowed concurrency. Whether the RP reader observes the state before or after the Delete, much of the reader can execute in parallel with the writer. This is not the case in the RWL implementation. The increased concurrency of the RP implementation allows it to scale better than the RWL implementation.

Listing 3.4 gives pseudo-code for a relativistic delete. The primary difference between the relativistic implementation and a standard implementation is the inclusion of `wait-for-readers` between where the node gets unlinked and where it gets freed. Although `rp-publish` is not required for ordering purposes for the delete, it is typically included for software engineering purposes. Rather than reasoning about which pointer updates require `rp-publish` for ordering and which do not, all reader-visible pointer updates are made through `rp-publish`. Another note about this pseudo-code, `rp-free` could be used to asynchronously free the memory for the deleted node. Doing so would allow the write to complete sooner because it would not include the `wait-for-readers`. The ordering requirements would still be met because `rp-free` includes an implied `wait-for-readers` prior to freeing the memory.

```
1   void delete_after(prev)
2   {
4      node = prev->next;
5      rp_publish(prev->next, node->next);
6      wait-for-readers();
7      free(node);
8   }
```

Listing 3.4: Pseudo-code for a relativistic delete.

3.2.3 Move Forward

The previous two sections showed how `rp-publish` and `wait-for-readers` can be used to control ordering. Their use was illustrated via linked list insert and delete operations. Both of these operations are fairly simple. They both involve two updates, but both of these updates are to the same node. The insert initialized and published a node. The delete unlinked and reclaimed a node. This section examines a slightly more complex example: a Move Forward operation in which a node is moved from one

location to a location later in the list. The primary difference between this example and the previous is that relativistic readers have the potential to see intermediate states.

Figure 3.5 shows a linked list where node B is moved from the second to the third position within the list. Three pointers need to be changed to effect this move: The pointers out of A, B, and C. Reader-writer locking uses mutual exclusion so readers never see intermediate states. As a result, it does not matter what order these pointers are updated. Readers can see the state before none of the pointers are updated or the state after all three are updated.

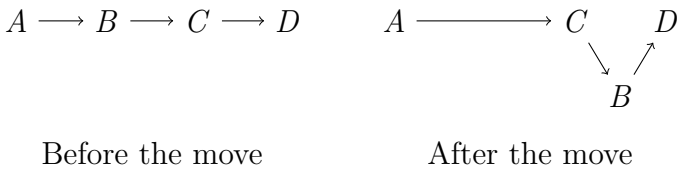


Figure 3.5: Moving a node to a location later the list.

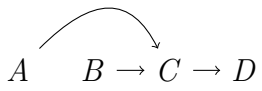
With relativistic programming, readers can see intermediate states. The possible reader-visible states are: the node is in its old position, the node is in its new position, the node is in both positions, and the node is in neither position. The "both" and "neither" options represent intermediate states. To decide which (if any) of these intermediate states are valid, we need correctness criteria. The relativistic programming correctness criteria depend on the ADT the list is representing.

Consider a Map where each node contains a ⟨key,value⟩ pair. With a list representing a Map, the location of a node within the list does not matter. As a result, the move does not change the state of the ADT so readers that observe the list at any intermediate point in the update should observe the same ADT state. (See Triplett et. al. [62] for an example of why an item may need to be moved in a list without affecting the state of the represented ADT.) Maps support lookups which either re-

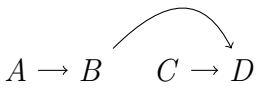
turn a value or `not-found`. If the lookup encountered the state where the node was not in either position, it could erroneously return `not-found`. However, if the lookup encountered the state where the node was in both positions, it would simply return the value from the first matching node it found. Because both positions contain the same value, it does not matter which node the value is taken from.

The following discussion is for a Map implemented with a relativistic linked list. Later in this section will be a discussion of a MultiMap which allows keys to be repeated with different values.

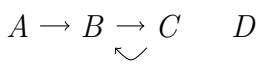
Hardware architectures do not have an operation that can update three pointers atomically. If the pointers are updated one at a time, an invalid intermediate step will be visible as shown in Figures 3.6a- 3.6c. Even a three-way atomic update would not be sufficient to prevent invalid states. A reader looking for C that was at B when the atomic update took place would fail to find C because after the update, C precedes B. The relativistic solution is to place a copy of B in the new location prior to removing B from its original location as shown in Figure 3.6d.



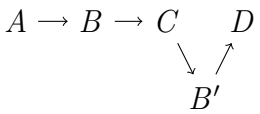
a) Change link out of A first: B is unreachable.



b) Change link out of B first: C is unreachable.



c) Change link out of C first: D is unreachable.



d) Insert a copy of B in the new location.

Figure 3.6: Possible intermediate states when moving node B.

Figure 3.7 shows the ordering relationships for this move. The event *deref(X)* means reading the contents of node X including following the pointer to the next node. The figure shows three readers: Reader 1 observes the list with B in its original position; Reader 2 observes the list with B in both positions; Reader 3 observes the list in its new position. The fourth option—observing B in neither position—is specifically prevented, so the three readers shown in the figure represent an exhaustive enumeration of the possibilities.

Readers 1 and 2 have *deref(A)* $\rightarrow$ *unlink(B)* so these readers observe B. Any reader that observes B has *deref(B)* $\Rightarrow$ *reclaim(B)*. The required-before relationship can be satisfied by placing a `wait-for-readers` between *unlink(B)* and *reclaim(B)*. This case is identical to removing a node from a linked list as discussed in Section 3.2.2.

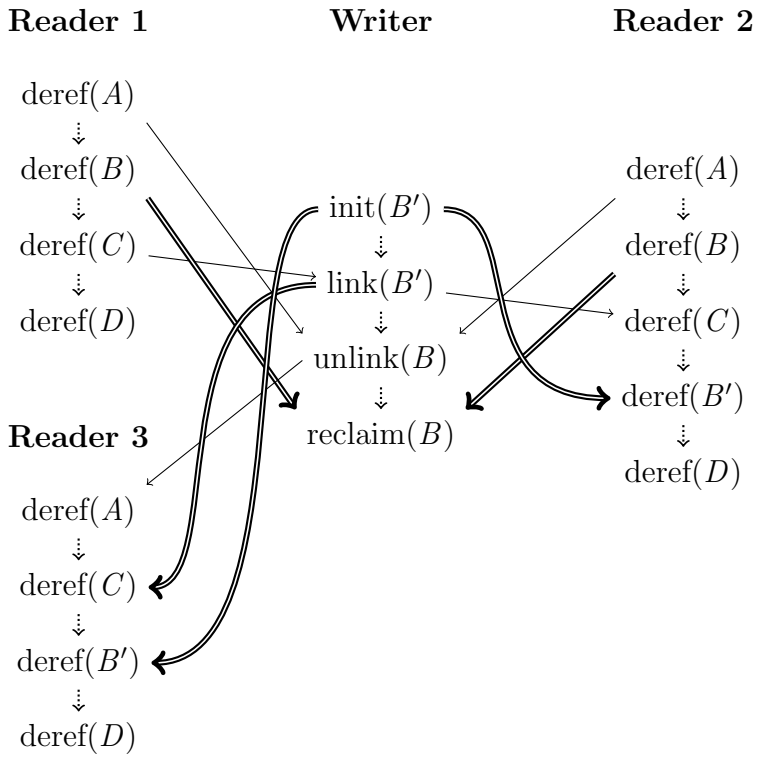


Figure 3.7: Ordering relationships for a reader and a writer performing a move.

Reader 2 has *link(B′)* $\rightarrow$ *deref(C)* and Reader 3 has *link(B′)* $\Rightarrow$ *deref(C)* so

these readers observe B'. Observing B' means there is a required-before relationship between *init(B′)* and *deref(B′)*. This relationship is satisfied using `rp-publish` to perform the *link(B′)* operation. This situation is the same as adding a node to a linked list as discussed in Section 3.2.1.

Reader 3 has *unlink(B)* $\rightarrow$ *deref(A)* meaning the reader did not observe B. If this reader is looking for the key in B, it must observe B' otherwise it will erroneously report that B is not in the list. The relationship *link(B′)* $\Rightarrow$ *deref(C)* captures this dependency. Because *link(B′)* and *unlink(B)* are both writes, and because Thread 3 observed the second of these two writes, the *dependency ordering* mechanism can be used to enforce *link(B′)* $\Rightarrow$ *deref(C)*. The `rp-publish` primitive guarantees that all writes in program order prior to the publish will be visible to readers before the published value. If `rp-publish` is used to *unlink(B)*, then any reader that observed the unlink must also observe any previous writes, namely, *link(B′)*. This is precisely the condition needed.

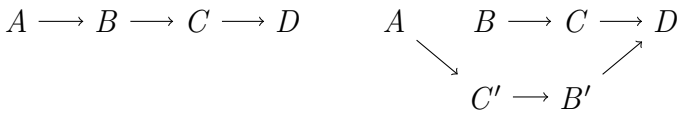
Listing 3.5 shows pseudo-code for a relativistic move forward. The parameters `new_pos` and `old_pos` represent the nodes immediately before the new and old locations of the node to be moved. All of the required-before relationships can be satisfied by using `wait-for-readers` prior to freeing the memory for `node` in line 10 and by using `rp-publish` to update the pointers in lines 7 and 8. As with the Delete operation, `rp-free` could be used to asynchronously free the memory for `node` so the update could complete earlier.

The preceding analysis was for a linked list that implemented a Map where duplicate nodes are assumed to represent the same item. If a linked list was used to represent a MultiMap, then duplicate nodes would be assumed to be two different items that happened to have the same key. In this case, a move that allowed a reader to see the node in both locations might be considered erroneous. A relativistic move

```
1   void move_forward(new_pos, old_pos)
2   {
4      node = old_pos->next;
5      copy = copy_node(node);
6      copy->next = new_pos->next;
7      rp_publish(new_pos->next, copy);
8      rp_publish(old_pos->next, node->next);
9      wait-for-readers();
10     free(node);
11  }
```

Listing 3.5: Pseudo-code for a relativistic move forward. The `new_pos` and `old_pos` parameters are the nodes prior to the new and old positions of the node to be moved.

that required readers to only see the state of the list before the move or after the move with no intermediate states would require copying all the nodes between the new and old location as shown in Figure 3.8. Readers in which *deref(A)* $\rightarrow$ *link(C′)* are guaranteed to see B but not B'. Readers for which *link(C′)* $\rightarrow$ *deref(A)* are guaranteed to see B' but not B.



Before the move After the move

Figure 3.8: Nodes B and C were copied so B can be moved without readers seeing intermediate states.

The Move Forward operation represents a departure from Insert and Delete because the RP implementation has the potential to allow readers to see intermediate states. Which states (if any) are visible to readers can be controlled by the programmer through copying nodes. While this is more complex than a reader-writer locking solution, the RP solution should be faster and more scalable both because it allows more concurrency and because the RP solution uses the lighter weight

dependency ordering mechanism to control ordering instead of the heavier weight *wait* mechanism.

3.2.4 Move Back

A Move Back consists of moving a node from later in the list to a location earlier in the list. While there are many similarities to a Move Forward, if the list represents a Map this move requires a `wait-for-readers` to preserve the correctness properties where Move Forward could be handled using the *dependency ordering* mechanism.

Figure 3.9 shows node B being moved from the third to the second position in a linked list. A copy of B (labeled B') is placed in the new location prior to removing the original. If the list represents a Map, then the correctness conditions for this move are as follows:

1. if *deref(C)* $\rightarrow$ *unlink(B)* then *deref(B)* $\Rightarrow$ *reclaim(B)*
2. if *deref(A)* $\rightarrow$ *link(B')* then *deref(C)* $\Rightarrow$ *unlink(B)*
3. if *link(B')* $\rightarrow$ *deref(A)* then *init(B')* $\Rightarrow$ *deref(B')*

The first condition is the same as for a Delete. The second condition guarantees that if B' is missed, B will be seen. The third condition is the same as for an Insert.

Figure 3.9: Node B is moved to an earlier location within the list.

Figure 3.10 shows the ordering relationships for a move back of node B and two readers. Both readers are looking for the key in B. In Reader 1, *link(B')* $\rightarrow$ *deref(A)*,

so the reader finds the key at B' and stops its search. Because this reader accessed B', the node must be initialized before being accessed. The relationship *init(B′)* ⇒ *deref(B′)* captures this dependency, and this dependency is the same as for inserting a node in a linked list as discussed in Section 3.2.1. This required-before relationship is guaranteed by using `rp-publish` to perform the *link(B′)*.

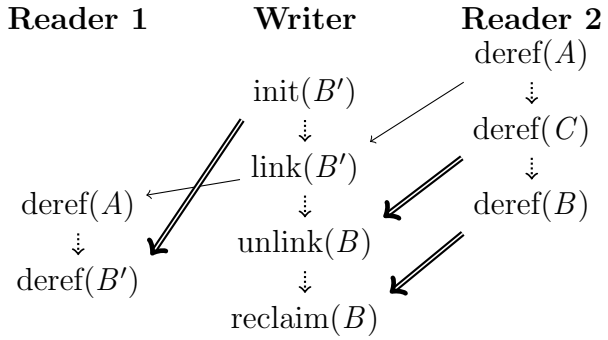


Figure 3.10: Ordering relationships for a reader and a writer performing a move back of B.

In Reader 2, *deref(A)* → *link(B′)* so this reader does not observe B' and therefore must observe B. The relationship *deref(C)* ⇒ *unlink(B)* captures this dependency. A memory barrier is not sufficient to enforce this relationship. If a reader has read past A prior to *link(B′)*, then a memory barrier cannot cause the reader to "back up" and observe B'. Similarly, a memory barrier attached to *unlink(B)* will not stall until the reader performs *deref(C)*. The solution is to place a `wait-for-readers` between *link(B′)* and *unlink(B)* as shown in Line 8 of Listing 3.6. The `wait-for-readers` will allow all readers for which *deref(A)* → *link(B′)* to complete prior to *unlink(B)* thus enforcing *deref(C)* ⇒ *unlink(B)*.

The final *required-before* relationship in Reader 2, which exists between *deref(B)* and *reclaim(B)*, can be satisfied by placing a `wait-for-readers` between *unlink(B)* and *reclaim(B)* as shown in Line 10 of Listing 3.6. This use of `wait-for-readers`

is the same as in removing a node from a linked list as discussed in Section 3.2.2. This second `wait-for-readers` can be taken out of the write path by using `rp-free` to asynchronously free the memory.

```
1   void move_back(new_pos, old_pos)
2   {
4      node = old_pos->next;
5      copy = copy_node(node);
6      copy->next = new_pos->next;
7      rp_publish(new_pos->next, copy);
8      wait-for-readers();
9      rp_publish(old_pos->next, node->next);
10     wait-for-readers();
11     free(node);
12  }
```

Listing 3.6: Pseudo-code for a relativistic move back. The `new_pos` and `old_pos` parameters are the nodes prior to the new and old positions of the node to be moved.

Much like with Move Forward, the RP approach to Move Back differs from an RWL implementation because intermediate states are visible. If the list represents a Map, both the RWL and RP approaches use the *wait* mechanism to control ordering. However, the RP approach still allows greater concurrency because portions of the read can overlap with the write. The greater concurrency should allow the RP approach to scale better than the RWL approach.

3.2.5 General Guidelines on Using RP Primitives

The previous sections gave examples of the use of `rp-publish` and `wait-for-readers`. This section gives general guidelines in their use. A technical report [61] gives a more complete treatment of these guidelines.

Nodes in relativistic data structures contain two types of data: immutable and mutable. Generally, the immutable data is the payload of the nodes and the mutable

data is the pointers used to connect the nodes. Writers must guarantee that, once a reader obtains a reference to a node, the immutable portion of the node will not change. If a writer needs to change an immutable portion of a node, it must make a copy of that node, change the node in its private memory, and then replace the old node with the new one by changing a mutable pointer. If multiple changes to the mutable data need to be made to appear atomic, the same copy-on-update procedure can be used.

There are two rules for the placement of the `rp-publish` and `wait-for-readers` barriers. The first one is simple and deals with `rp-publish`. Any time reader-visible mutable data is changed, it must be changed using `rp-publish`. (Mutable data is reader-visible if there is a reader-visible link to the node containing the mutable data.) Doing so guarantees that all changes to mutable data from a single writer will be seen in order by all readers.

The second rule deals with `wait-for-readers`. If readers have a traversal order (e.g. from the root of a tree to a leaf), then if two causally related changes to mutable data are made in the same direction as the traversal, they must be separated by a `wait-for-readers`. An example of this is the Move Back described in Section 3.2.4. The Move Back required two changes to mutable data: The copy of the moved node was placed early in the list then the moved node was removed from later in the list. These changes are causally related and they occurred in traversal order so a `wait-for-readers` was required between them.

In the Move Forward example described in Section 3.2.3 there were two changes. The first one involved placing a copy of the moved node later in the list and then removing the original from earlier in the list. Because these operations happen in the reverse of traversal order, `wait-for-readers` was not needed between them.

The use of `wait-for-readers` for safe memory reclamation fits the traversal order

rule. There are two changes in a reclamation. The first one is to remove the node from the data structure, the second is the reclamation of the node's memory. Removing the node from the data structure involves changing the pointer that points at the node. This pointer must be earlier in the traversal order than the contents of the node that get reclaimed, so these two changes happen in traversal order and thus need a `wait-for-readers` between them.

There are two rules for readers: `rp-read` must be used when reading mutable data and each mutable data item must only be read once. The `rp-read` serves two purposes: it forces the compiler to issue a load instruction (instead of re-using a previously loaded value) and it enforces dependent read consistency. These actions are the same as for the C++0x `memory_order_consume` load.

If a mutable data item is read multiple times, the value of that item might change between each read. Listing 3.7 illustrates what might happen if a mutable item is read more than once. The code snippet shows a condition for terminating a linked list traversal. The pointer `node->next` may change between the condition in Line 1 and the return statement in Line 2. If this happens, the data returned will not come from the same node that was examined.

```
if (node->next->key == key) {
    return node->next->value;
}
```

Listing 3.7: Invalid coding pattern because `node->next` may change between lines 1 and 2

3.3 Considering Multiple Writes

The previous sections examined the ordering constraints between readers and a single writer. There are other considerations when considering a reader and multiple writers.

These considerations are examined in this section.

The timeline in Figure 3.11 shows a writer performing a delete and two readers. Using the terminology from Section 3.2.2 the diamond on the write line marks the *unlink*. The diamonds on the read lines mark the *deref*. Both readers begin before the writer and in both cases *deref* $\rightarrow$ *unlink*. Reader R1 finishes its read-section prior to the *unlink*. This case is indistinguishable from a reader-writer locking reader that executed just prior to the write. But Reader R2 continues well after the writer completed its operation. This leaves open the possibility that Reader R2 will be concurrent with another write operation and it may see the results of the subsequent write without observing the results of the first one.

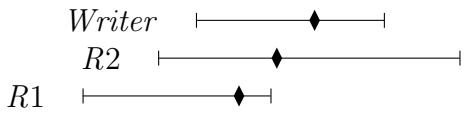


Figure 3.11: Timeline showing reader $R1$ that completes before the write takes effect and $R2$ that observes the state of the data structure prior to the write, but extends well after the write.

Figure 3.12 shows a reader concurrent with two writers. The relativistic correctness criteria requires that a reader concurrent with a writer can see the state immediately prior to the write or immediately after the write, but no other state. The case presented in Figure 3.12 requires a clarification of this rule. A reader concurrent with multiple writes can see the state prior to any of the writes, or the state generated by any combination of the concurrent writes being applied to the initial state. If I is the initial state, then the reader in Figure 3.12 can see one of the states $I, I + W1, I + W2, I + W1 + W2$.

If the second write is dependent on the first, then observing $W2$ without observing $W1$ could potentially lead to an invalid observation. To prevent this from happening,

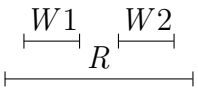


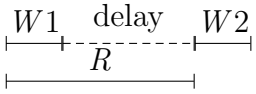
Figure 3.12: Operation R can see operations W1 and W2 in any order.

a `wait-for-readers` could be inserted between the two writes (for example by calling `wait-for-readers` prior to releasing the lock used for mutual exclusion between writers). The presence of the `wait-for-readers` would guarantee that no reader was concurrent with both $W1$ and $W2$. Any reader concurrent with $W2$ would be guaranteed to see the state after $W1$. Note that this is the same as the traversal order rule mentioned in Section 3.2.5. In order for a reader to see $W2$ but not $W1$, $W1$ must have been early in the traversal order and the reader read past that location before $W1$ took effect. $W2$ must have been later in the traversal order and the read took effect between $W1$ and $W2$.

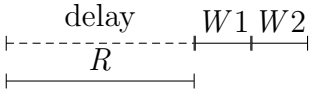
The state observed by the reader is specific to that reader. If Figure 3.12 had shown two readers both concurrent with $W1$ and $W2$, then each reader could have observed different states. One could have observed $I + W1$ and the other $I + W2$. If this were so, the two readers would disagree on the order of the updates. The first reader would claim $W1$ happened first (because it saw $W1$ but not $W2$). The second reader would claim $W2$ happened first. The order of observations is local to (or relative to) a particular reader. There is no "global observer" that defines the correct order. Each reader is allowed to observe writes in their own reference frame. Stated another way, in timelines, such that in Figure 3.11, there are ordering relationships between the readers and the writer, but there are no ordering relationships between the readers. The readers are drawn on the same time scale for graphical convenience, but because neither reader observes the behavior of the other, nothing can be said about the relative orders of the two readers.

The potential reordering is limited in scope to the length of any concurrent reader. If there are two writes A and B such that there is no reader concurrent with both A and B, then all readers will agree on the order of A and B—that is to say, A and B will be totally ordered with respect to all readers.

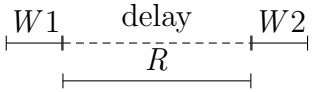
Reader-writer locking provides a total order of writes by delaying certain operations through mutual exclusion. If a relativistic solution requires a total order on writes, then `wait-for-readers` can be used to delay writes to preserve this total ordering. The delays to totally ordered relativistic writes are no worse than the delays imposed by mutual exclusion in reader-writer locking. This is shown in Figure 3.13 (see also [48]). For both relativistic programming and reader-writer locking, the delay can be up to the length of a reader. If the reader-writer implementation favors readers, then the delay can be any arbitrary length—the writer could starve. With relativistic programming writer starvation cannot happen.



a) Relativistic approach



b) Reader-Writer approach where R goes first



c) Reader-Writer approach where $W1$ goes first

Figure 3.13: Timelines showing different delays. In a) $W2$ is delayed because $W1$ holds the write-lock until R completes. In b) $W1$ is delayed because R acquires the reader-writer lock first. In c) $W1$ acquires the lock first but $W2$ is still delayed for the full duration of R. The relativistic version suffers the minimum delay.

Read-sections and write-sections do not just happen spontaneously. They happen in response to external or internal events. When considering timing and ordering, it is useful to not only look at the timing and ordering of the read-sections and write-sections, but also the timing in relation to the events that triggered the read-sections and write-sections. Figure 3.14 shows the timing of two readers and a writer. The circles show the external events that trigger the reading and writing. The diamonds show the time when the writer updated a pointer and when the readers observed that pointer.

In the relativistic programming approach, reads are allowed to run concurrently with the write. Reader R1 observes the data structure prior to the update. Reader R2 observes the data structure after the update. This is evident by the order of the diamonds.

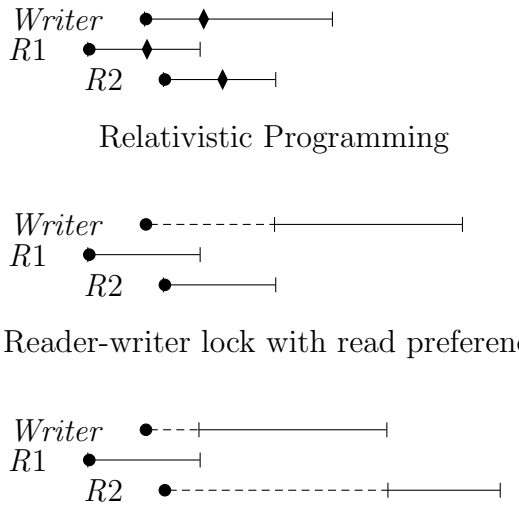


Figure 3.14: Timing of a writer and two readers using the relativistic approach and reader-writer locks with read preference and write preference or fair. The relativistic approach allows both the earliest possible writes and earliest possible reads.

The reader-writer lock read preference case shows how the write is delayed until

all readers have finished. The delay is shown by the dashed line. In this case, both readers observe the data structure prior to the update. Note that this violates the order of the events that triggered the read and write operations. The reader-writer lock write preference or fair case shows a delay in both the writer and reader R2. In this case, the readers make the same observations as in the relativistic case, but the writer and one of the readers are delayed. This type of delay has the potential to significantly reduce scalability.

The reader-writer locking approach obtains consistency by delaying operations. However, this delay has the potential to cause operations to be ordered differently than the events that triggered these operations. Relativistic programming on the other hand allows the earliest possible completion of operations.

3.4 Quantitative Evaluation of RP vs. RWL

This section examines three aspects of performance: 1) the reasons why relativistic primitives can yield better performance, 2) the problem with small critical sections for reader-writer locks, and 3) the write delay for totally ordered relativistic programs compared with the write delay for reader-writer locking programs. This section is not intended to show the overall performance benefits of relativistic programming. Chapter 4 will show the superior performance of a relativistic red-black tree. Other publications [62, 63] have made the case that other relativistic algorithms are both high performance and highly scalable.

3.4.1 Analysis

Reader-writer locking primitives are necessarily slow because they require two-way communication between the primitives in different threads. Figure 3.15 shows the ordering relationships that implement the reader-writer locking semantics. The required-

before relationship must be implemented using the *wait* mechanism because a memory barrier is not sufficient to cause the locking thread to wait until the lock is released. The *wait* mechanism requires that the *unlock* write a value that the *lock* waits for. But in order for the *lock* to know that it needs to wait for the *unlock*, the previous *lock* must have also written a value.

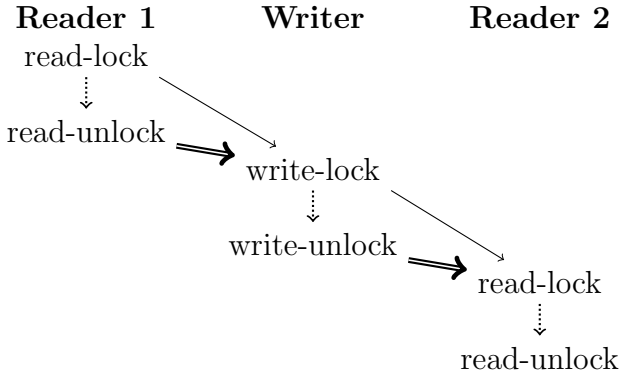


Figure 3.15: Orderings required for reader-writer lock semantics.

The pseudo-code in Listing 3.8 shows a potential reader-writer lock implementation. This is not intended to be a complete implementation because it does not handle multiple readers. However, it shows the necessary writes and waiting reads. The waiting read and the following write must happen as an atomic unit; otherwise another thread may acquire the lock between the read and the write. Atomic read-modify-write instructions are typically used to combine read and write as an atomic unit. Atomic read-modify-write instructions are as much as several orders of magnitude slower than regular instructions [9], and they also serialize the writes. The combination of these two properties mean the locking primitives can form a significant serial bottleneck that can limit scalability even for read-only workloads.

In contrast to RWL, the communication between relativistic primitives is asymmetric: Writers performing a `wait-for-readers` need to know about the existence of

Read Primitives

```
read_lock() {
  while (!writing)
  {}

  reading = TRUE;
}

read_unlock() {
  reading = FALSE;
}
```

Write Primitives

```
write_lock() {
  while (!reading)
  {}

  writing = TRUE;
}

write_unlock() {
  writing = FALSE;
}
```

Listing 3.8: Possible implementation of reader-writer lock primitives.

readers, but readers never need to know about the existence of writers. In Figure 3.16 the writer can observe the existence of Reader 1 because *start-read*$_{R1}$ $\rightarrow$ *start-wait*. This occurred-before relationship requires *end-read*$_{R1}$ $\Rightarrow$ *end-wait*. But Reader 2, which can observe the existence of the writer, has no further dependencies—in other words, Reader 2 does not care about the existence of the writer. This asymmetry allows the read-side primitives to be implemented without atomic read-modify-write instructions.

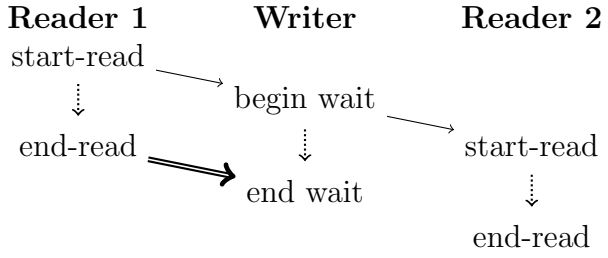


Figure 3.16: Orderings required for relativistic programming semantics.

The pseudo-code in Listing 3.9 shows a potential implementation of relativistic primitives. There is a global Epoch counter which gets incremented on each call to `wait-for-readers`. There is an array of epochs, one per reader. When a reader starts,

it stores the global epoch into its per-reader epoch. When the reader finishes, it clears its per-reader epoch. Because only one reader writes to each location in the per-reader epoch list, no synchronization is required for these writes. The `wait-for-readers` waits for each reader to either not be reading or to be reading in an epoch newer than the one being waited on. No atomic read-modify-write instructions are required in the read-path so relativistic programs should scale for read-only (or read-mostly) workloads even for small critical sections.

Read Primitives

```
start-read() {
  Reading[reader] = Epoch;
}

end-read() {
  Reading[reader] = 0;
}
```

Write Primitives

```
wait-for-readers() {
  int my_epoch;

  atomic_inc(Epoch);
  my_epoch = Epoch;

  for (readers) {
    while (Reading[reader] != 0 &&
        Reading[reader] < my_epoch)
    {}
  }
}
```

Listing 3.9: Possible implementation of relativistic programming primitives.

3.4.2 Scalability as a function of critical section size

The previous section made the claim that RWL readers will not scale for sufficiently small critical sections because of the cost and serial nature of the atomic read-modify-write instructions used to update a common lock variable. Because relativistic programing readers do not need atomic read-modify-write operations, nor do they need a common lock variable, they should scale for any size critical section. This claim was tested using a linked list micro benchmark. The list contained sorted data, and

operations included lookups, inserts, and deletes. The average length of the critical section can be varied by changing the average length of the linked list. The tests were run on a four processor quad-core Intel Xeon system (16 hardware threads) running Linux 2.6.32. The relativistic programming implementation was supplied by Usermode RCU developed by Desnoyers et. al. [17, 16]. Appendix A gives additional information on the data collection methodology. The following synchronization mechanisms were benchmarked:

nolock No synchronization was used. This is valid for read-only work loads but could lead to data corruption in the presence of updates. *nolock* is included as a potential upper bound on performance.

rp Relativistic programming using the Usermode RCU implementation. The *write-lock* is a pthread mutex.

torp Totally Ordered Relativistic programming using the Usermode RCU implementation. *torp* is the same as *rp* except that `wait-for-readers` is called before releasing the write-lock.

rwlr Reader-writer locks that favor readers. The implementation is based on Mellor-Crummey and Scott [50].

rwlw Reader-writer lock that favors writers. The implementation is based on Mellor-Crummey and Scott [50].

Figure 3.17 shows read performance for a linked list of size one—the shortest meaningful critical section. Both *nolock* and *rp* scale linearly. There is overhead associated with the *rp* read-side primitives, so the performance of *rp* is less than that of *nolock*; however, the overhead is both small and fixed—additional threads do not increase the overhead. Neither of the reader-writer locks scale because additional

threads mean more contention for the lock variable. The additional contention for the lock variable exceeded any benefit of parallelism so the best performance was with a single thread.

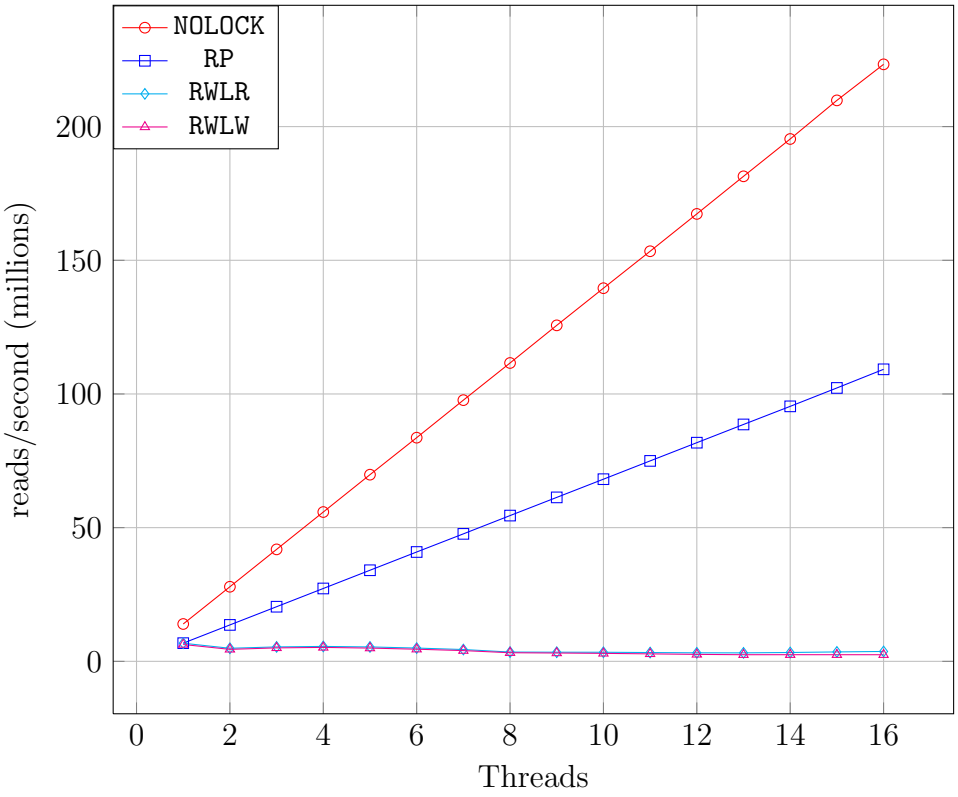


Figure 3.17: Read performance with a list size of 1. Reader-writer locks do not scale because of the short critical sections. Note that the *RWLR* and *RWLW* are on top of each other.

Figure 3.18 shows read performance with a list of size 1,000. All synchronization mechanisms show good scalability because the critical section is large with respect to the cost of synchronization. However, even for lists of size 1,000, the performance of *rwlw* (and to a lesser extent *rwlr*) is noticeably less than linear for higher thread counts. The effects of lock contention can be seen even for critical sections that involve thousands of instructions.

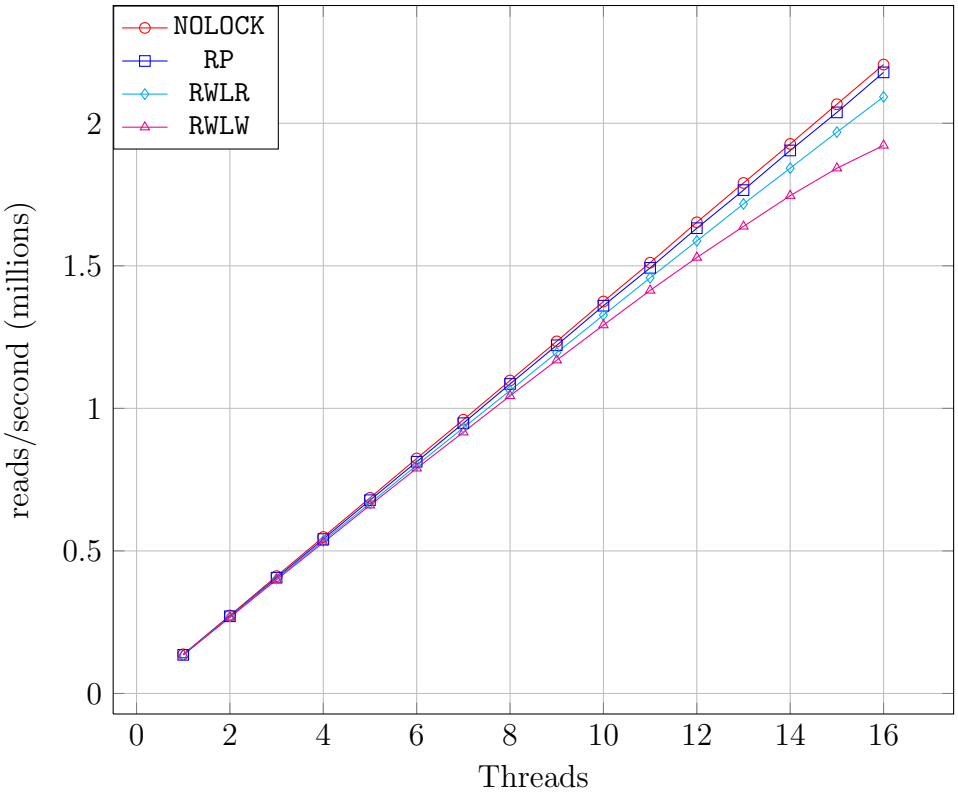


Figure 3.18: Read performance with a list size of 1000. All synchronization mechanisms scale because the size of the critical section is large with respect to the cost of synchronization.

3.4.3 Comparing RP, TORP and RWL

Totally Ordered Relativistic Programming (*torp*) delays writes so no reader is concurrent with two writes. The expected delay is the length of the average read-section. Section 3.3 claimed that the delay in releasing the lock for *torp* was no worse than the delay for acquiring the lock in *rwl*. To test this, an experiment was run with a single update thread and 15 read threads. This configuration generated the maximum amount of read-contention for the writer, and should cause the largest read-generated delays. The experiment was repeated over a large range of list sizes. Figure 3.19 presents the results of this experiment and they confirm that *torp* and *rwl* have similar delays. In particular, the update performance of *torp* is better than *rwlw* for small

list sizes. For larger list sizes, *rwlr* is slightly better than *torp*. The reason is that the *torp* implementation used for this benchmark required extra fields in the structure to allow memory to be reclaimed asynchronously. The additional data causes additional pressure on the caches and memory bus. A *torp* implementation that did not require this additional data should continue to out perform *rwlw* for all list sizes.

Notice that *torp* has significantly better read performance than *rwlw*. The additional reads should increase contention on the writes, but despite this, *torp* still out performs *rwlw* on the write side.

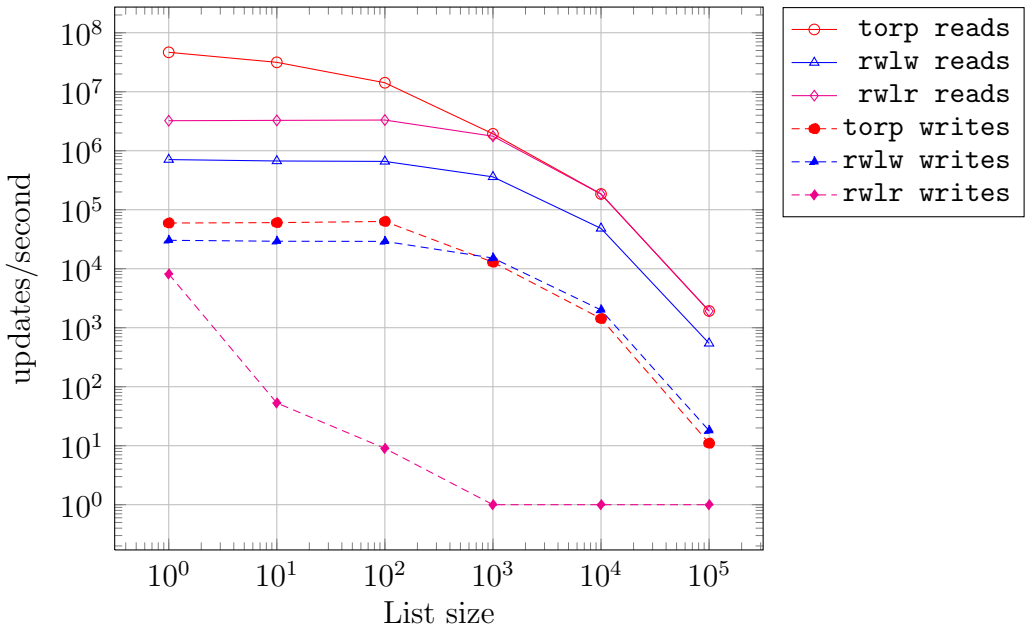


Figure 3.19: Performance of an update thread concurrent with 15 readers. The performance of *torp* and *rwlw* are similar because *torp* has to wait for all current readers before releasing the write lock and *rwlw* has to wait for all current readers before acquiring the write lock.

3.5 Summary

This chapter introduced the ordering constraints imposed by relativistic programming. The two relativistic programming primitives, `rp-publish` and `wait-for-readers`

can be used by relativistic writers to limit the orderings that are visible to concurrent readers. The rules for using these primitives are simple and easy to understand. When these primitives are used correctly, only orderings that yield correct reads are allowed. The burden for correctness is placed on writers so readers can proceed without the need for heavy weight synchronization. The different ordering requirements of relativistic programs allows them to have higher performance and better scalability than their reader-writer locking counterparts.

Chapter 4

Supporting Complex Multi-write Updates

To date, RP algorithms have dealt with fairly simple data structures and updates have mostly involved single nodes. This chapter presents the implementation of a relativistic red-black tree. Implementing a relativistic red-black tree presents a significant challenge because some updates involve as many as three nodes. Hardware does not support an atomic operation that can be used to perform the entire update at once—and, as shown in Section 3.2.3, even if hardware did support such an operation, that would not be sufficient to prevent readers from seeing invalid states. As a result, an algorithm must be developed that keeps the tree in an always-valid state even in the midst of these multi-node updates[1].

4.1 Introduction

Red-black trees (RBTrees) are used to store sorted ⟨key,value⟩ pairs. They guarantee $O(\log(N))$ performance for lookup, insert, and delete operations. They also have relatively low cost for mainaining the balance properties of the tree. RBTrees are used extensively in operating systems. For example, in the Linux kernel they are used for I/O schedulers, the process scheduler, the ext3 file system, and in many

[1]Herlihy [31] suggested a method for creating concurrent data objects where the entire object needs to be copied. This method would allow complex updates, but it is ruled out due to poor performance.

other places [41]. RBTrees are also used by many applications and are provided by class libraries such as Java.util.TreeMap [1].

Unfortunately, RBTrees are difficult to parallelize. Most implementations use a single global lock to protect concurrent accesses. Coarse grained locking prevents these implementations from scaling because accesses are serialized by the lock. Because accesses can be easily divided into reads (single lookups and complete traversals) and writes (inserts and deletes), a reader-writer lock can be used. This approach scales for some number of read threads, but eventually the contention for the lock dominates and the approach no longer scales (see the performance data in Section 4.4.1 for evidence of this behavior).

Fine grained locking of red-black trees is problematic. Because updates may affect all the nodes from where the update occurred back to the root, the simplest approach of acquiring a write lock on all nodes that might change degrades to coarse grain locking because all updaters must acquire a write lock on the root. If one attempts to acquire write locks only on the nodes that will actually be changed, it is difficult to avoid deadlock. Because updates always involve a leaf (see Section 4.3), one approach to acquiring locks would be to start with the leaf to be changed and acquire locks from the bottom up. However such a scheme may collide with a reader acquiring read-locks on the way down the tree. If instead, the highest node to be modified as part of rebalancing was identified, then the locks were acquired top down, an overlapping update may win the race and modify the tree in such a way that the analysis that determined which nodes need to be modified is no longer valid.

Transactional Memory (TM) approaches provide a more automatic approach to disjoint access concurrency. Many researchers use an RBTree as one of the benchmarks for their Software Transactional Memory (STM) systems. Some of these show good scalability [20, 19], however, they tend to compare their STM implementation

against other STM implementations. This masks the fact that overall performance suffers because of the high overhead of all STM systems. Dragojevic et al. [18] compared STM's against a sequential implementation and found that for RBTrees it often took four to eight cores to equal the performance of a single core non-TM solution. With some implementations, they found that performance was worse than a sequential implementation over the full range of processors available on a given system.

Bronson et al. [10] developed a concurrent AVL tree[2]. Their approach allows readers to proceed without locks, but the readers have to check each step of the way to see if the tree has changed or is in the process of changing. If so, the reader has to wait and retry. Because readers do not acquire locks, fine grained locking of the writers is less problematic. Their approach is quite complicated and this degrades performance as more code must execute at each node of the tree. The RP approach allows over twice the read performance and up to three times the single threaded update performance (see Section 4.4 for details). Their approach allows concurrent scalable updates but performance is no better than for a more straight forward STM implementation (see Chapter 6 for evidence of this).

In order to accelerate update operations, a number of researchers have attempted to decouple rebalancing from insert and delete [29, 30]. This allows updates to proceed more quickly because individual inserts and deletes do not have to rebalance the tree. The rebalancing work can potentially be done in parallel and some redundant work can be skipped. None of this improves read access time, and readers and writers still need some form of performance limiting synchronization between them.

This chapter presents a relativistic RBTree implementation that has low-overhead, wait-free, linearly-scalable read performance out to at least 64 hardware threads even in the presence of a concurrent update. Although relativistic programming is not yet

[2]AVL trees are similar to red-black trees, but they have a different balance property.

a fully general technique, the development of a relativistic data structure as complex as a concurrent RBTree is a significant milestone because it shows that relativistic programming can be applied to complex data structures that require simultaneous multi-node updates.

4.2 A Relativistic Red-Black Tree Algorithm

Because red-black trees are well known and well documented [7, 29, 55, 58], a complete explanation is not repeated here. Rather, a brief overview is presented simply to facilitate a discussion of the relativistic implementation. In particular, the individual steps that make up red-black tree algorithms are discussed without discussing the glue that combines these steps. This approach is taken because the glue is not impacted by the relativistic implementation.

Red-black trees are partially balanced, sorted, binary trees. The trees store ⟨key,value⟩ pairs. They support the following operations:

insert(key, value) inserts a new ⟨key,value⟩ pair into the tree.

lookup(key) returns the value associated with a key.

delete(key) removes a ⟨key,value⟩ pair from the tree.

first()/last() returns the first (lowest keyed) / last (highest keyed) value in the tree.

next()/prev() returns the next/previous value in key-sorted order from the tree. Some implementations of these primitives pass arguments specifying the last accessed node.

Red-black trees are sorted by preserving the following properties:

1. All nodes on the left branch of a subtree have a key less than the key of the root of the subtree.

2. All nodes on the right branch of a subtree have a key greater than or equal to the key of the root of the subtree.

The tree is balanced by assigning a color to each node (red or black) and preserving the following properties:

1. Both children of a red node are black.

2. The black depth of every leaf is the same. The black depth is the number of black nodes encountered on the path from the root to the leaf.

These invariants are sufficient to guarantee $\mathrm{O}(\log(N))$ lookups because the longest possible path (alternating black and red nodes) is at most twice the shortest possible path (all black nodes). The operations required to rebalance a tree following an insert or delete are limited to the path from the inserted/deleted node back to the root. The rebalancing is most often $\mathrm{O}(1)$ but in the worst case is $\mathrm{O}(\log(N))$. This means that inserts and deletes, including rebalancing, can also be done in $\mathrm{O}(\log(N))$.

An insert or delete may violate the balance properties. If so, the tree is rebalanced by recoloring nodes or performing restructure operations, sometimes called rotations. Restructures always involve three adjacent nodes: child, parent, and grandparent. See Figure 4.1 for an illustration of the two types of restructure operations. Following an insert, at most one restructure is required to restore balance. Following a delete, at most two restructures are required [29].

The correctness criteria proposed in Section 2.3 requires that the data structure be a representation of an ADT. The relativistic RBTree presented here is appropriate for a *Map* where each key is unique. This implimentation is not suitable for a *Multimap*

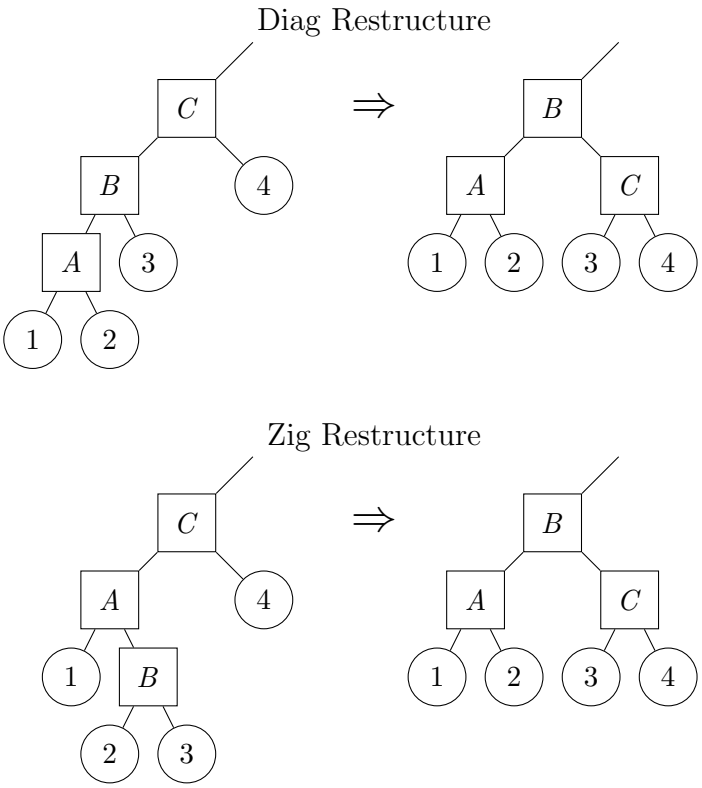


Figure 4.1: Restructure operations used to rebalance a red-black tree. There are left and right versions of these, but they are symmetric so only the left version is shown here.

where a key can be repeated with different values. Applying the correctness definition to a Map yields the following properties:

1. Lookups will always find a key that exists in the Map.

2. Traversals will always return values in the correct order without skipping any values that exist in the Map.

Because reads have a duration, and because updates can proceed concurrent with reads, it is possible that the Map will change during a read. As a result, the phrase "nodes that exist in the Map" does not have a static definition. The dynamic definition is given below using the following definitions:

$\rightarrow$ occurred-before in the sense that Lamport [40] uses *precedes* so that if $A \rightarrow B$ then every event in A preceeds every event in B. Note that in Chapter 3, $\rightarrow$ was applied to operations that were atomic (reads and writes of memory). In this chapter, $\rightarrow$ is applied to non-atomic operations; otherwise its meaning is similar.

R_K a read looking for key K or a complete traversal of the Map.

I_K the insert of key K.

D_K the delete of key K.

E_x the effect of operation x. The effect of I_K is that node K exists in the Map. The effect of D_K is that K no longer exists in the Map. Effects are assumed to take effect instantaneously so that even if two operations are concurrent, there can be an occurred-before relation on their effects (though relativistic programming does not require all threads to agree on the order of the effects).

For simplicity, assume that the Map starts empty and ends empty and a given key value is only inserted once. Then "nodes that exist in the Map" is defined as follows:

1. If $I_K \rightarrow R_K$ and $R_K \rightarrow D_K$ then K exists in the Map and must be observed by R_K in the correct traversal order.

2. If $R_K \rightarrow I_K$ or if $D_K \rightarrow R_K$ then K does not exist in the Map and must not be observed by R_K.

3. If I_K is concurrent with R_K then K may or may not be observed by R_K depending on whether the relative view of R_K is $E_I \rightarrow E_R$ or $E_R \rightarrow E_I$.

4. If D_K is concurrent with R_K then K may or may not be observed by R_K depending on whether the relative view of R_K is $E_R \rightarrow E_D$ or $E_D \rightarrow E_R$.

Another way to state these properties is as follows: Properties one and two state that any update that strictly precedes a read must be observable by the read, and any update that strictly follows a read must not be observable by the read. Properties three and four state that any update that is concurrent with the read may or may not be observable by the read.

The following observations can be made about readers performing a lookup (for traversals, see Chapter 5):

1. Readers ignore the color of nodes.

2. Readers do not access the parent pointers in nodes.

3. Temporarily having the same item in the tree multiple times will not affect lookups provided all copies are in correct sort order within the tree. A positive result will return the first copy encountered (and because they are duplicates, it does not matter which copy is returned). A negative result (item not in tree) will return "not found" even if other keys are duplicated in the tree.

The implications of these observations are that updaters can change the color and parent pointers without affecting readers; updaters can also temporarily allow duplicates provided both duplicates are in valid sort order locations.

The following definitions will be used in the explanation of the tree operations:

internal node A node with two non-empty children.

leaf A node with at least one empty child.

Both internal nodes and leaf nodes contain ⟨key-value⟩ pairs.

Observe that if `next()` is called on any internal node, the result is always a leaf. This is true because `next()` on an internal node is the left-most node of the right subtree.

Given the above observations, the following can be said about the steps in an update:

Insert New nodes are always inserted at the bottom of the tree. This is possible because if `prev(new-node)` is any node with a non-null right branch, then from the observation above, the new node must be a leaf. If `prev(new-node)` had a null right branch, then the new node will be a leaf placed on that null right branch. A concurrent reader will either see the new node or not depending on whether `rp-publish` of the insert happens-before the `rp-read` of the pointer changed by `rp-publish`. But concurrent readers will never see an invalid state.

The insert may leave the tree unbalanced. If so, restructures or recolors (see below) are required to restore the balance properties of the tree.

Delete Nodes are only deleted from the bottom of the tree (possibly as part of a swap—see below). Similar to insert, a concurrent reader will either see the deleted node or not depending on the order of operations, but a reader will never see an invalid state. The memory for the deleted node must not be reclaimed while concurrent readers have a reference to it. Using the `rp-free` primitive will ensure that the proper delay occurs before the memory is reclaimed.

The delete may leave the tree unbalanced. If so, restructures or recolors (see below) are required to restore the balance properties of the tree.

Swap If an interior node needs to be deleted, it is first swapped with the next node in sort order (`= next(deleted-node)`) prior to removal. This makes the node to be deleted a leaf. Because a concurrent reader searching for the swapped node might be at a point in the tree between

the swapped node's new and old positions, special handling is required to ensure that such a reader sees the swapped node. The swapped node exists in the tree and therefore it must be observable in correct traversal order.

Restructures Restructure operations are used to rebalance the tree. Much like swap, restructures involve moving nodes. This requires special handling to keep the tree in an always-valid state.

Recolor Nodes get recolored as part of the rebalancing process. Recoloring does not involve changing the structure of the tree, only the colors applied to particular nodes. Because readers ignore the color of nodes, recoloring does not affect the read-validity of the tree.

The two operations that require special handling in a relativistic implementation are Swap and Restructure. These are described in greater detail below.

4.2.1 Swap algorithms

Two swap algorithms are presented below. The first one is the general algorithm that works in the general case. There is an optimized special case that applies when the swap node is the right child of the node to be deleted.

General Swap

Consider the delete of node B shown in Figure 4.2. Because B is an internal node, B will be swapped with C ($= next(B)$) prior to deletion. The new location of C is higher in the tree, so the swap qualifies as a Move Back operation as described in Section 3.2.4. Move Back operations require a `wait-for-readers` to maintain read-validity.

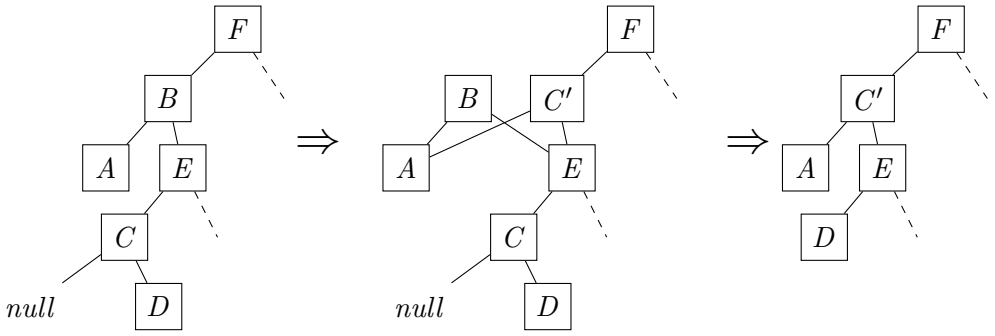


Figure 4.2: Tree before and after deletion of node B including one intermediate step

Rather than performing separate swap and delete steps, the two are combined as a single step as shown in Listing 4.1. C' is a copy of C. C' is changed so that it has the same color as B and the same children as B, but the key and data values of C.

The new node C' is linked into the tree in place of B. At this point, the value C is in the tree twice: once at C and once at C'. Any readers looking for C can be divided into two groups: those above C' will find the value at C', those at or below B will find the value at C. In either case, the correct value will be found. However, if the old node C is removed, any readers looking for the value C that were at or below B would miss the value. To avoid this problem, the updater calls `wait-for-readers` before removing C from the tree. This ensures that any readers at or below B will complete their read prior to C being removed. Any new readers will see C' and thus will not need to find C.

This algorithm differs from a non-RP algorithm in the following ways:

1. A copy of node C is placed in B's position rather than node C itself.

2. `rp-publish` is used to make reader visible pointer assignments to guarantee that changes to a node are visible before the node itself is reachable.

3. `rp-free` is used to release memory to ensure that no readers have references to

```
C = next(B);
C_prime = C.copy();

C_prime.color = B.color;

C_prime.left = B.left;
C_prime.left.parent = C_prime;

C_prime.right = B.right;
C_prime.right.parent = C_prime;

F = B.parent;
C_prime.parent = F;

if (F.left == B)
  rp-publish(F.left, C_prime);
else
  rp-publish(F.right, C_prime);

rp-free(B);

wait-for-readers();

E = C.parent;
rp-publish(E.left, C.right);
C.right.parent = E;

rp-free(C);
```

Listing 4.1: Code for swap

the memory when it is released.

4. A `wait-for-readers` is included so that no readers will miss seeing node C.

Special case: swap node is child of B

In the tree shown in Figure 4.3, C is $next(B)$. It also happens to be the right child of B. This represents a special case because no new nodes need to be created. The changes are made as shown in Listing 4.2. C takes the color of B. The left child of B becomes the left child of C. The node A now appears in the tree twice (once below B and once below C). Any reader encountering the tree in this state will find A regardless of where they were in their traversal when the changes were made. B is removed from the tree by linking C into the tree in its place. B is then freed asynchronously by calling `rp-free`.

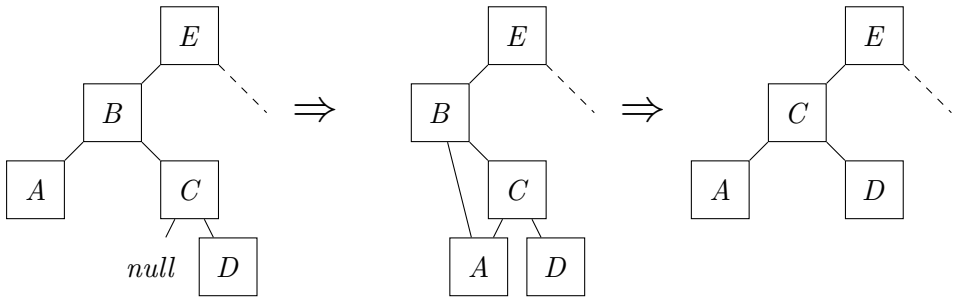


Figure 4.3: Tree before and after deletion of node B including one intermediate step

This algorithm differs from a non-RP algorithm only in the use of `rp-publish` to make pointer assignments, and `rp-free` to release memory.

4.2.2 Restructure

There are two cases for restructures depending on whether the three nodes involved form a diagonal or a "zig". Each of these can be further classified depending on

```
C = next(B);

C.color = B.color;
C.left = B.left;
C.left.parent = C;

E = B.parent;
if (E.left == B)
  rp-publish(E.left, C);
else
  rp-publish(E.right, C);

rp-free(B);
```

Listing 4.2: Code for special case Swap

whether it is left or right, but the left and right cases are symmetric, so only the left case will be described here.

Diag Left

Figure 4.4 shows a subtree with three nodes labeled A, B, C that need to be rotated so that B becomes the root of the subtree. The change places C lower in the tree, so this is a Move Forward and it can be made without a `wait-for-readers`. The changes are made as shown in Listing 4.3.

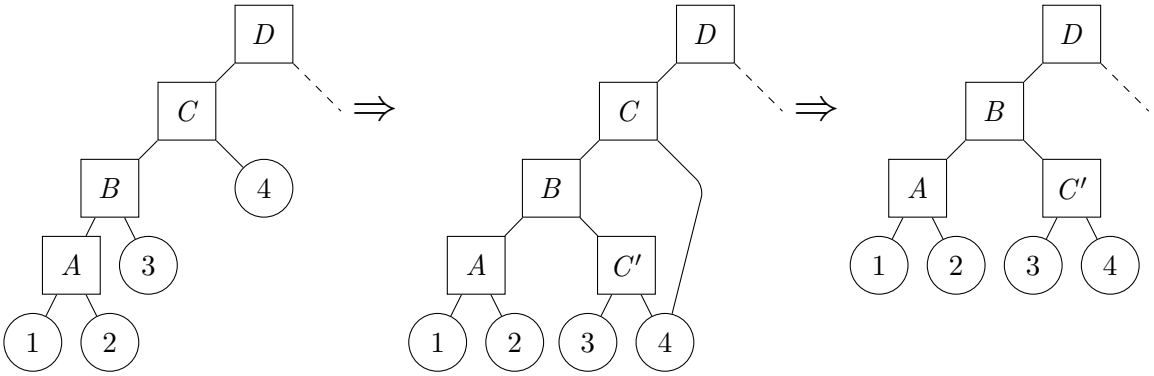


Figure 4.4: Arrangement of nodes before and after a diag restructure including one intermediate step.

C' is a copy of node C. The right child of B becomes the left child of C'. C' is then linked into the tree as the right child of B. At this point, the value C is in the tree twice. This is similar to swap in Section 4.2.1. However, in this case, the copy is placed lower in the tree rather than higher in the tree. As a result, the original node C can be removed without waiting for readers. Any readers between C and C' will still see C' even after C is removed from the tree.

```
1  C_prime = C.copy();
   C_prime.left = B.right;
   C_prime.left.parent = C_prime;

5  rp-publish(B.right, C_prime);
   C_prime.parent = B;

   D = C.parent;

10 if (D.left == C)
     rp-publish(D.left, B);
   else
     rp-publish(D.right, B);

15 B.parent = D;

   rp-free(C);
```

Listing 4.3: Code for diag left restructure

This algorithm differs from a non-RP algorithm as follows: a copy of a node was made rather than changing a node in place, and RP primitives were used for pointer assignment and memory reclaimation.

Zig Left

Figure 4.5 shows a subtree with three nodes labeled A, B, C that needs to be rotated so that B is the root of the subtree (this is known as a double rotation [29]). There are two ways to accomplish this: either a copy of B can be placed above A and C, or

copies of A and C can be placed below B. Because the first method involves moving the copy up in the tree, it requires a `wait-for-readers`. Even though the second method requires two copies, performance data showed that the second method is faster, so that method is described here (see Listing 4.4). In essence, the cost of a `wait-for-readers` was more than the cost of the second node copy which is not surprising because `wait-for-readers` is a fairly heavyweight primitive.

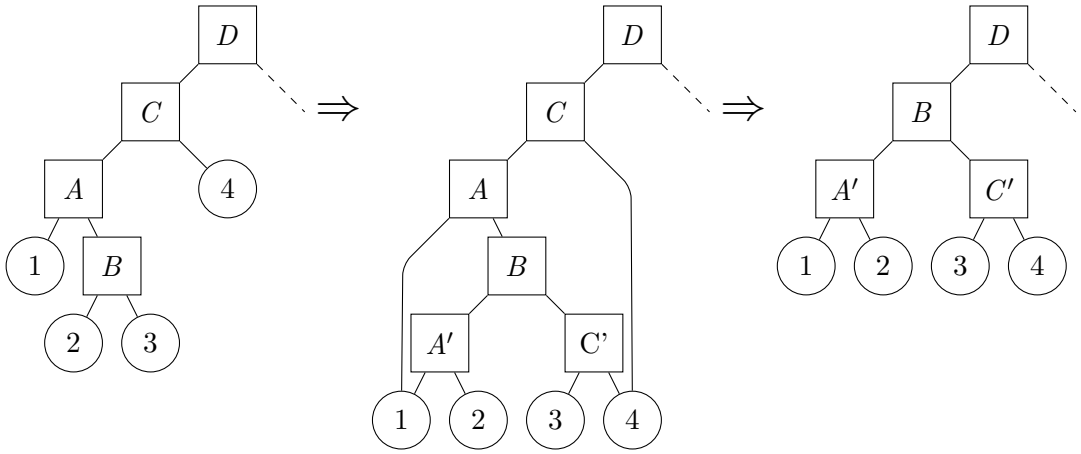


Figure 4.5: Arrangement of nodes before and after a zig restructure including one intermediate step.

A' is a copy of A. The left child of B becomes the right child of A'. A' is linked into the tree as B's left child. At this point, the value A appears in the tree twice. Because the new copy is placed below the original, this is a Move Forward operation so there is no need for a `wait-for-readers` before removing the original from the tree.

C' is a copy of C. The right child of B becomes the left child of C'. C' is linked into the tree as B's right child. The original nodes A and C are removed from the tree by making B a child of D.

This algorithm differs from a non-RP algorithm as follows: copies of nodes were made rather than changing nodes in place, and RP primitives were used for pointer assignment and memory reclamation.

```
A_prime = A.copy();
A_prime.right = B.left;
A_prime.right.parent = A_prime;

rp-publish(B.left, A_prime);
A_prime.parent = B;

C_prime = C.copy();
C_prime.left = B.right;
C_prime.left.parent = C_prime;

rp-publish(B.right, C_prime);
C_prime.parent = B;

D = C.parent;
if (D.left == C)
  rp-publish(D.left, B);
else
  rp-publish(D.right, B);

rp-free(A);
rp-free(C);
```

Listing 4.4: Code for zig left restructure

4.3 Linearizability

Linearizability [36, 65, 31] is a widely used correctness criteria for concurrent programs. An algorithm is linearizable if each operation takes effect through a single atomic operation. This section argues that the lookup, insert, and delete operations meet this criteria so that these operations are linearizable.

The arguments for the linearizability of these operations will be based on the following properties of the relativistic implementation:

Immutable nodes The key and value within a node never change once the node is created.

Sort The update methods preserve the sort property of trees such that all nodes on the left branch are less than the current node and all nodes on the right branch are greater than or equal to the current node.

Structural integrity The update methods never move a node in such a way that a lookup will fail to find that node.

The following subsections examine each of the steps of the relativistic tree algorithm and explain why they are linearizable. In particular, the lookup, insert, and delete operations have a single atomic operation where the operation takes effect, and the rebalancing operations do not violate this property.

4.3.1 Lookup

A lookup proceeds by examining the key in a node of the tree (initially the root of the tree), and taking one of three courses of action based on the key in the node:

1. The correct node has been found and will be returned

2. the lookup will proceed down the left branch

3. the lookup will proceed down the right branch

The `rp-read` that is used to transition to the last node examined on a lookup is the linearization point. Because `rp-read` involves reading a pointer, and because this operation is atomic on all modern hardware, this is a valid linearization point. The following reasons justify the choice of this `rp-read` as the linearization point:

The *immutable nodes* property means that even if the key is examined several times (e.g. once to determine if the lookup has found the node in question and again to decide which branch to look down) the answer will be the same each time. The *sort* property means that the correct branch will always be searched. The *structural integrity* property means that a node below a reader will never be moved above the reader in such a way that the reader will not see that node. As a result, the last `rp-read` in a lookup determines the result of the lookup provided the updates preserve the *sort* and *structural integrity* properties. The next sections show that these properties are preserved.

4.3.2 Insert

Nodes are always inserted at the bottom of the tree. The `rp-publish` that makes the new node visible to readers serves as the linearization point for an insert. The `rp-publish` involves writing a pointer, which is atomic on all modern hardware. Prior to the `rp-publish` the node does not exist in the tree and following the `rp-publish` it does. Provided the restructure operation (described below) preserves the *sort* and *structural integrity* properties, the restructure will not invalidate the `rp-publish` as the linearization point.

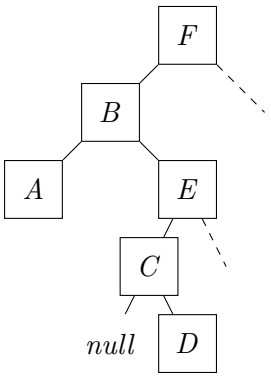
4.3.3 Delete

Leaf nodes are deleted with a single `rp-publish`. The `rp-publish` has the parent point to the non-null branch of the leaf node (if it exists) instead of the leaf node. If there is no non-null branch, the `rp-publish` has the parent point to `null`. The `rp-publish` serves as the linearization point of the delete. Prior to the `rp-publish` the node exists in the tree and following the `rp-publish` it does not. Provided the restructure operation (described below) preserves the *sort* and *structural integrity* properties, the restructure will not invalidate the `rp-publish` as the linearization point.

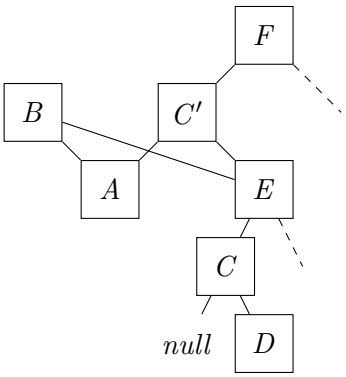
4.3.4 Swap

When an interior node is deleted, it is first swapped with the next node in sort order (the left most node on the right branch). Figure 4.6 enumerates the reader visible states in this process. The first reader visible change is when node C' gets linked into the tree in place of node B using `rp-publish`. This `rp-publish` is the linearization point for the removal of B as can be seen by considering lookups for node B. The linearization point for the lookup is the `rp-read` of `F.left`. If the `rp-read` happens before the `rp-publish` the lookup will find B otherwise it will report that B does not exist in the tree. The `rp-publish` preserves both the *sort* and *structural integrity* properties.

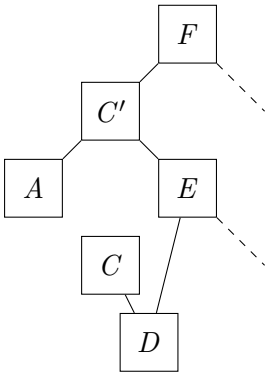
The second reader visible change is when C is removed from the tree by linking D in its place using `rp-publish` to update `E.left`. The combination of the two changes has the effect of moving C to a new location within the tree in possible violation of the *structural integrity* property. However, a `wait-for-readers` occurs between the changes so that any reader at B or E looking for C will find C before it is removed. Any reader looking for C that began after the `wait-for-readers` will find C' so it will terminate its search before encountering node E.



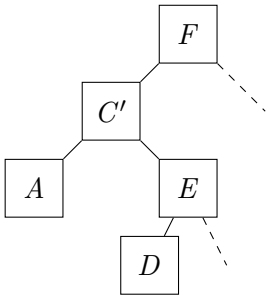
a) Initial subtree



b) After insertion of C'



c) After removal of C



d) Final subtree

Figure 4.6: All reader visible states in the deletion of node B through a swap

After C is removed from the tree, the node still exists in memory because concurrent readers may still have a reference to it. The final change takes place when C is reclaimed. C is reclaimed after all current readers have finished so that no readers have a reference to C when it is reclaimed.

4.3.5 Restructure

Restructures come in both the diag and zig forms. Restructures happen only after inserts, deletes, or swap. The linearization point for the update occurred in the insert, delete, or swap, so it only needs to be shown that the restructure operations do not violate the *sort* or *structural integrity* properties. Let's consider the diag restructure first.

Figure 4.7 shows the reader visible states of a diag restructure operation. The first change places C' below B. This temporarily violates the *sort* property because node 4 appears on both the left and right branches of C. However, lookups for all nodes on the subtree rooted at C will terminate correctly. In particular, a reader at B will find node 4 through C'. Any readers at or above C will find 4 as the right branch of C. Any readers looking for C will stop at C so the fact that there is a duplicate on the left branch of C will not cause any problems. So even though the *sort* property is violated, the violation will not cause any lookups to fail. The second change corrects the temporary violation of the *sort* property.

The second change removes C from the tree leaving C'. The effect of the first two changes is that C is moved to a new location within the tree. However, after each step, any lookups will complete correctly regardless of where in the tree the reader was when the change happened. Thus the *structural integrity* property is preserved.

After C is removed from the tree, the node still exists in memory because concurrent readers may still have a reference to it. The final change takes place when C is

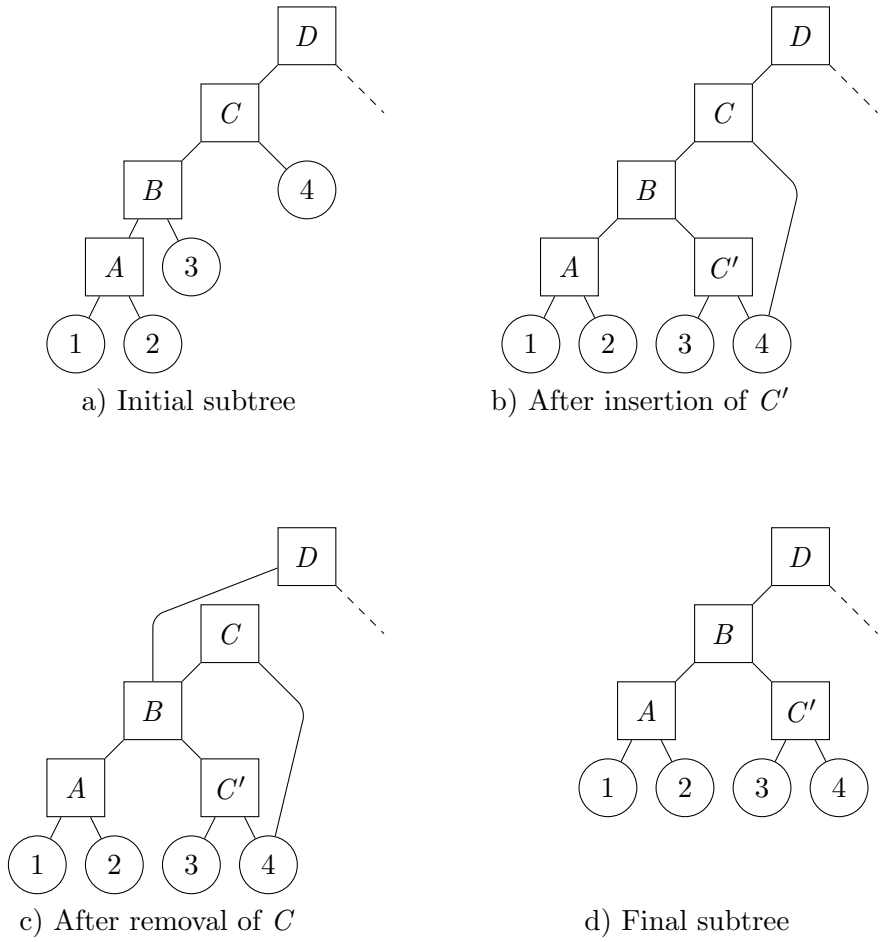


Figure 4.7: All of the reader visible states in a diag restructure

reclaimed, whichs happens after all current readers have finished so that no readers have a reference to C when it is reclaimed.

Figure 4.8 shows the reader visible states of a zig restructure. The first change is inserting A' below B. This change does not violate either the *sort* or *structural integrity* properties. The second change inserts C' below B. This change temporarily violates the *sort* property because node 4 is on both the left and right branch of C. As with the diag restructure, all lookups within the subtree will complete correctly. The third change corrects the temporary violation of the *sort* property by removing both A and C. The *structural integrity* property is not violated by this change because A and C

continue to be visible to all readers that could potentially be looking for these nodes.

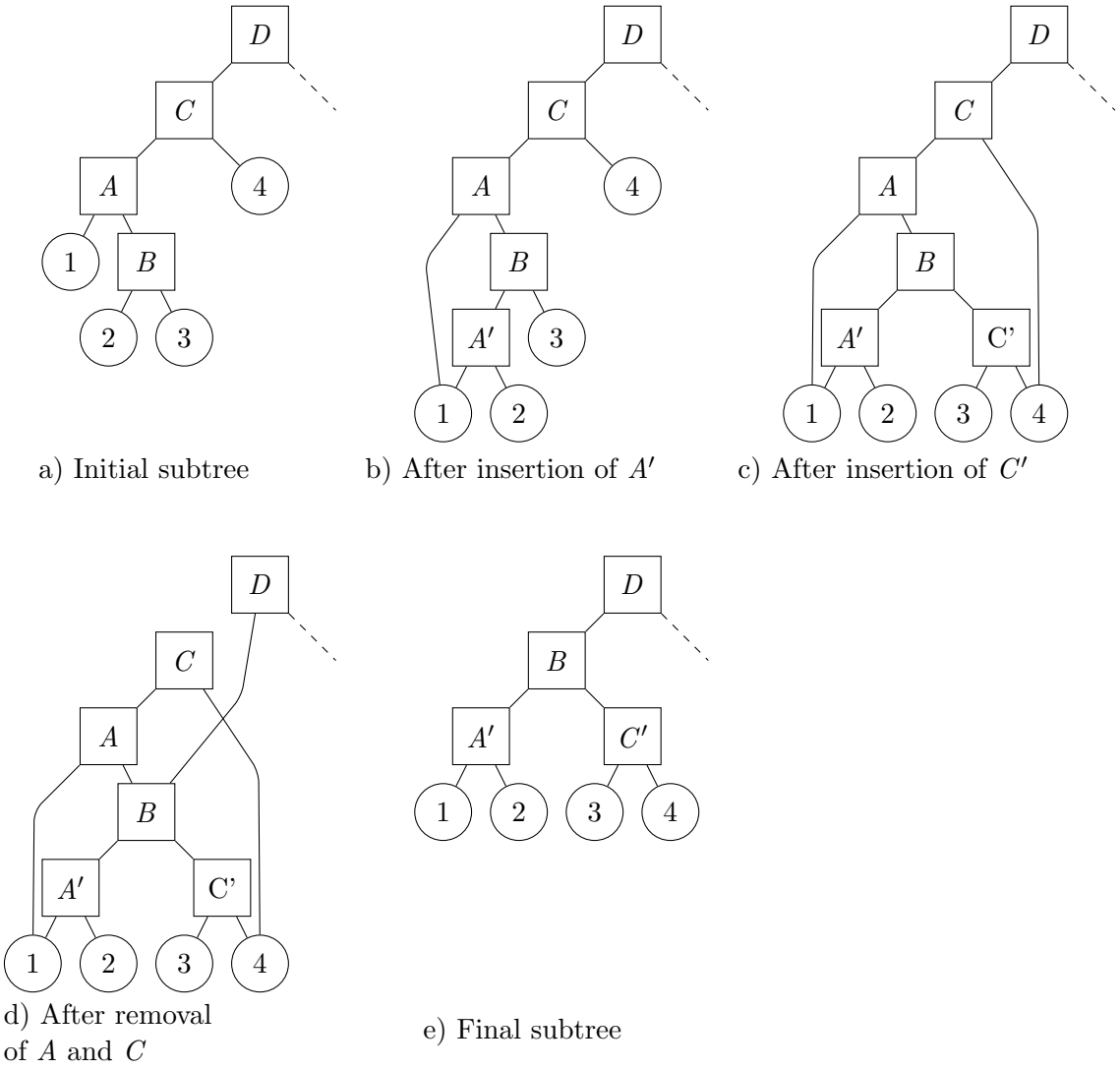


Figure 4.8: Reader visible states for a zig restructure

After A and C are removed from the tree, the nodes still exist in memory and concurrent readers may still have a reference to them. The final change takes place when A and C are reclaimed, which happens after all current readers have finished so that no readers have a reference to these nodes when they are reclaimed.

4.3.6 Summary of linearizability arguments

Relativistic programming does not guarantee a total order on all events. This relaxed ordering is part of what allows the higher performance of relativistic algorithms. However, this does not mean that ordering is totally chaotic. Readers are allowed to proceed almost unsynchronized because the writers guarantee the necessary ordering properties through the use of `wait-for-readers`. In the case of the swap algorithm, the `wait-for-readers` was used to preserve the *structural integrity* property while moving node C higher in the tree. Not only are RP algorithms not chaotic, it was shown that the lookup, insert, and delete operations are fully linearizable with each other. Note that the linearizability of these operations is a comment on these specific operations, not a comment on the linearizability of relativistic implementations in general. Chapter 5 will examine the linearizability of complete traversals, and will show that there are correct, but non-linearizable relativistic traversal algorithms.

4.4 Performance

Performance data was collected using the following synchronization techniques:

`nolock` No synchronization was used. This is not a valid concurrent implementation because it leads to data corruption. It was tested as a theoretical upper bound (highest performance, but not a data safe implementation) to compare the other algorithms against.

`lock` A pthread mutex was used and shared between readers and writers. There was no parallelism while accessing the tree.

`rwlr` A reader-writer lock that favors readers. The implementation was derived from Mellor-Crummey and Scott [50].

`rwlw` A reader-writer lock that favors writers. The implementation was derived from Mellor-Crummey and Scott [50].

`rp` This is the relativistic implementation described in this chapter.

`ccavl` The concurrent AVL implementation by Bronson et al. [10]

Note 1: all the RBTree algorithms except `rp` used a "standard" red-black tree implementation that did not perform copy-on-update. The standard algorithm was used to avoid biasing the results against the non-RP implementations because copy-on-update is slower than the standard implementation.

Note 2: Bronson's `ccavl` algorithm was originally developed in Java and it relied on Java's garbage collection mechanism. The algorithm was ported to C to match the implementation of the other algorithms. However, no garbage collection was provided for their algorithm. As a result, the performance numbers reported here for `ccavl` are better than what a complete implementation would provide because the numbers presented here do not include the cost of garbage collection.

The test created a tree and preloaded it to a given size with a random set of values. Threads were created to perform operations on the tree (lookups, inserts, and deletes). The threads were allowed to run for a fixed period of time and the total number of operations performed was reported.

Threads were of two types: readers and updaters. Readers performed lookups for values in the tree. Updaters removed a value from the tree and then inserted a different value. By pairing deletes and inserts, the size of the tree remained fixed.

Tests were performed on trees of size 64 and 64K nodes. The choice of sizes was somewhat arbitrary, but represented a small tree where the probability that multiple threads were accessing the same node concurrently was high, and a large tree where the probability of multiple threads accessing the same node concurrently was low.

The behavior at these two sizes was sufficiently similar that exploring other tree sizes was not deemed profitable[3]. The graphs for trees of size 64K are presented here, and comments indicate where there were differences in the size 64 graphs.

Two sets of benchmarks were run. In the first set, all threads were readers. In the second, one thread was an updater and the rest were readers. The second set shows how concurrent reads affect update performance. Multiple writers are considered in Chapter 6.

Performance data was collected on a Sun UltraSPARC T2 running SunOS 5.10. The UltraSPARC T2 has eight cores each supporting eight hardware threads for a total of 64 hardware threads. Appendix A gives additional information on the data collection hardware and methodology.

4.4.1 Read Performance

Figure 4.9 shows the read performance of the red-black tree. The performance of `rwlw` and `lock` were strictly worse than `rwlr`, so they were left off the figure for clarity. The reads here are individual lookups, not complete traversals. For complete traversals, see Section 5.3. The following observations can be made from the figure:

1. `rp` read performance scales linearly to at least 64 threads.

2. `rp` read performance approaches unsynchronized performance. `rp` values were approximately 93% of the `nolock` values.

The `ccavl` algorithm scales linearly to at least 64 threads. However, its performance was only about 20% that of `rp`. When reporting the performance of an

[3]The RP algorithm is intended to scale with the number of processors. No comment is made on its ability to scale with tree size. Scaling with tree size is more a function of the memory and cache architecture of the machine than it is of the synchronization mechanism. As a result, it was not necessary to benchmark very large tree sizes.

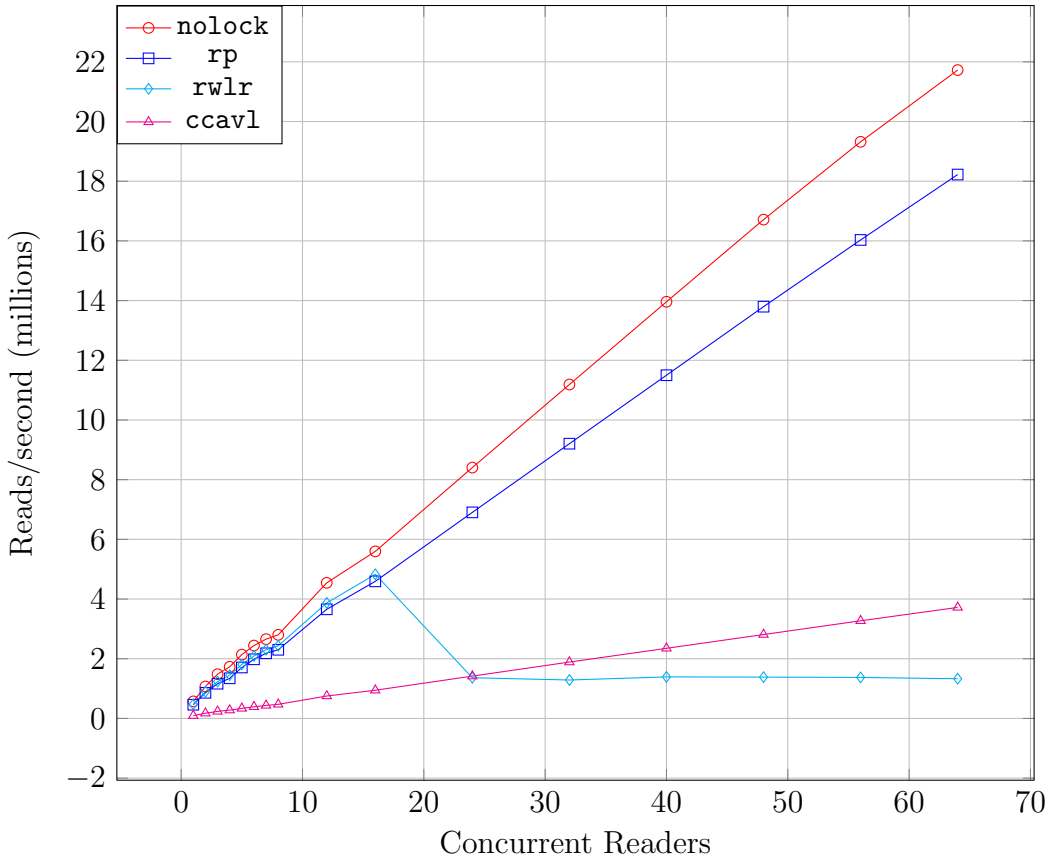


Figure 4.9: Read performance of 64K node red-black tree using a variety of synchronization techniques.

algorithm, it is important to report both scalability and absolute performance. Relativistic programming does well on both counts.

When `rp` read performance is plotted with and without a concurrent updater, the two lines are indistinguishable as shown in Figure 4.10. The average difference in `rp` read performance with and without a concurrent updater was less than 1%. This shows that a concurrent updater does not impact read performance. For `ccavl`, with large tree sizes the contended and uncontended read performance is very similar because the `ccavl` algorithm takes advantage of disjoint access parallelism and with sufficiently large trees, most accesses are disjoint. Figure 4.10 shows performance for a tree of 64 nodes. With this size tree, enough of the writes are accessing the

same portion of the tree as the reads that read performance suffers. For `rwlw`, at low thread counts, Figure 4.10 clearly shows that the writes interfere with read performance. At higher thread counts, contention for the lock variable is sufficient that read performance suffers with or without a concurrent writer.

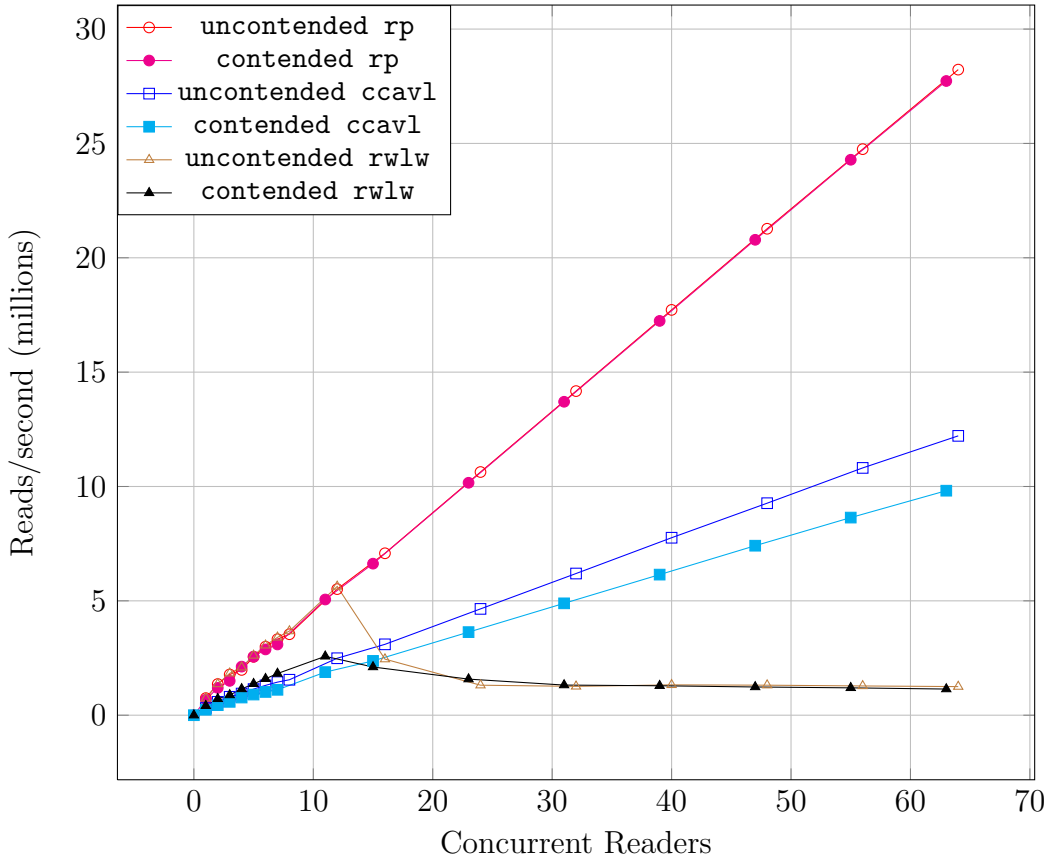


Figure 4.10: Contended and uncontended read performance of 64 node red-black tree using a variety of synchronization techniques.

4.4.2 Update Performance

Figure 4.11 shows the performance of a single update thread running concurrently with a varying number of read threads. The X axis shows the number of read threads. The leftmost data point on each line (zero read threads) shows the uncontended update performance. The remainder of the data points show how the update perfor-

mance is affected by a varying number of readers.

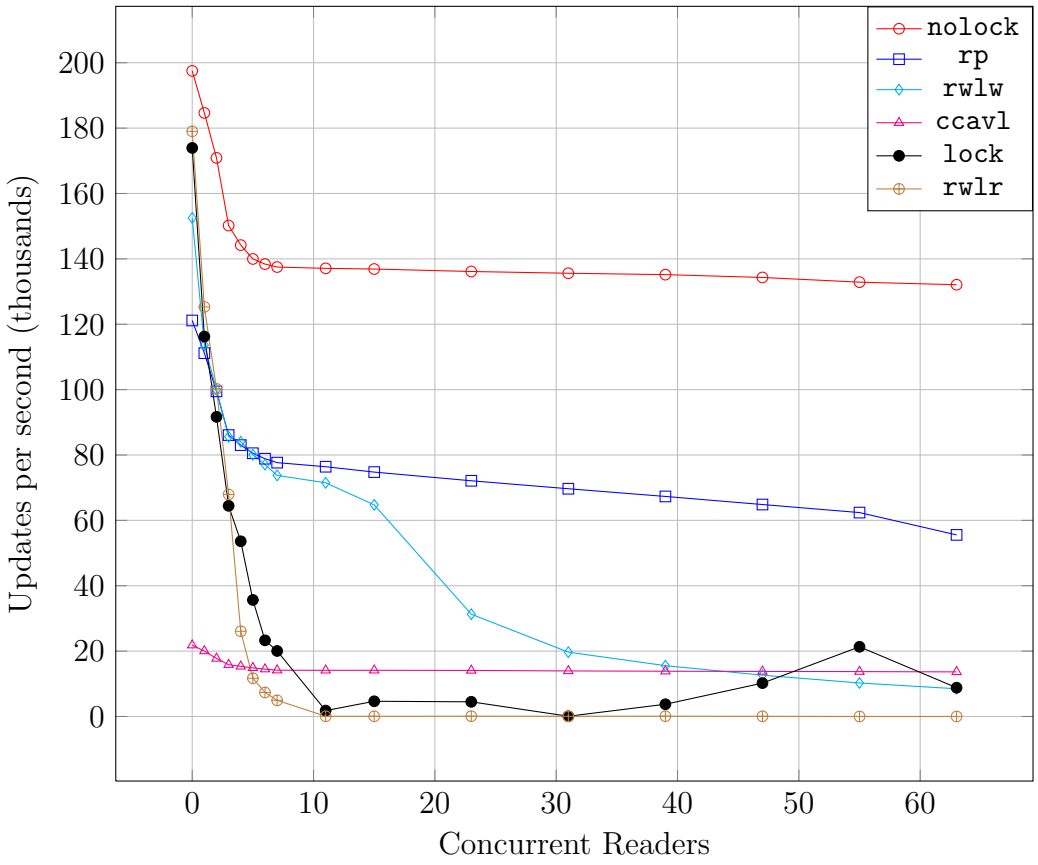


Figure 4.11: Update performance of 64K node red-black tree with a single updater and multiple readers. The left-most data point shows uncontended write performance. The remainder of the data points show the update performance with a variable number of concurrent readers.

Of all the synchronization methods, `ccavl` had the worst uncontended update performance. The second worst was `rp` although it still measured 93% of `rwlw` performance (`rwlw` was the best of the other valid methods). With three or more concurrent readers, `rp` update performance is better than any of the other synchronization methods (with the exception of `nolock` which leads to data corruption). As the number of concurrent readers increases, the advantage of `rp` is more pronounced. With a smaller tree, it takes more concurrent readers to give a clear advantage to `rp` because the higher read-rate of the `rp` implementation caused greater contention on the

updater with smaller trees. With a tree size of 64 nodes, `rp` had better write-side performance if there were 6 or more concurrent readers.

4.5 Summary

This work has made the following contributions to relativistic programming:

1. Demonstrated that relativistic programming can be applied to a complex data structure—a balanced binary tree—that requires multi-node updates.

2. Showed how `wait-for-readers` can be used to order the read-visibility of writes. Prior to this work, the primary use of `wait-for-readers` was to delay the reclamation of memory.

3. Provided a red-black tree algorithm that allows for high performance, linearly-scalable reads, even in the presence of concurrent updates.

Chapter 5

Supporting Complex Read Patterns

The previous chapter showed that relativistic programming can be applied to complex data structures that require multi-node updates. However, the reads considered in Chapter 4 were simple lookups; another read pattern for RBTrees is complete traversals where every node is accessed in sort order. Complete traversals present problems to a relativistic implementation for a variety of reasons: Traversals violate some of the assumptions on which the relativistic implementation was based; traversals take a long time which can cause excessive delays in `wait-for-readers`; and the relativistic linearization point does not apply to traversals.

This chapter explores some of the issues raised by relativistic readers that have long durations (such as tree traversals). The chapter is outlined as follows: Section 5.1 outlines how traversals present different problems compared with single lookups. Section 5.2 presents three different approaches to solving these problems, and Section 5.3 shows the performance trade-offs for the three approaches.

5.1 The Problem with Traversals

This section describes three potential problems with traversals with a relativistic RBTree: the potential for invalid traversals, long duration `wait-for-readers`'s that delays writes, and non-linearizable traversals.

5.1.1 Invalid Traversals

An in-order-traversal is used to access all the nodes in an RBTree in their sorted order. An in-order-traversal is often defined recursively as shown in Listing 5.1. This algorithm depends upon the program stack to maintain state about which nodes have been visited and which are yet to come. A non-recursive version of this algorithm could be developed that used a stack data structure instead of the program stack to maintain this same state.

```
1  in_order(node)
   {
       if (node == NULL) return;

5      in_order(node.left);
       process(node);
       in_order(node.right);
   }
```

Listing 5.1: Recursive in-order-traversal implementation

If an update occurred during a traversal that used the recursive algorithm, the update could leave the tree in a state inconsistent with the data on the stack. Consider the restructure shown in Figure 5.1. A traversal at node B prior to the restructure would have C on the stack. If the restructure happened at this point, the traversal would visit node C when following the right branch of B and then again when it pulls C off the stack after processing B thus C would be processed multiple times. To avoid this type of problem, concurrent updates must not alter the portion of the tree structure that is represented by the contents of the stack.

Another traversal algorithm makes use of parent pointers instead of using a stack to store state. The basic principle is that if a node is the right child of its parent, then the parent has already been processed. Listing 5.2 gives a traversal algorithm based on this principle. The `leftmost(node)` primitive returns the leftmost leaf of the

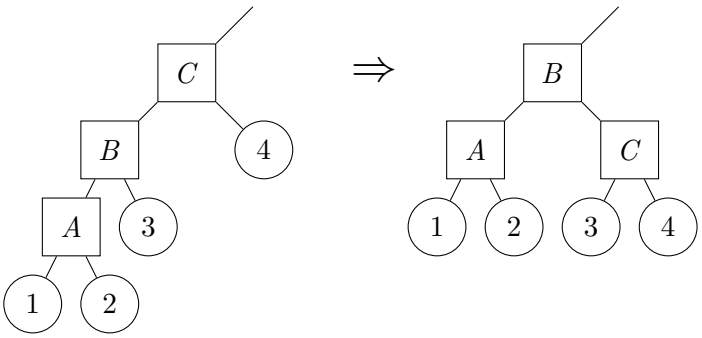


Figure 5.1: A traversal at node B when this restructure happened would visit node C twice.

subtree rooted at `node`. The traversal starts in Line 3 by finding the leftmost leaf of the tree. The outer while loop (Line 5) processes each node. If a node has a non-null right branch, the right branch is followed in Line 9. If a node has a null right branch, the traversal moves up in the tree. The while loop in Line 12 continues moving up as long as the current node is the right child of its parent.

Figure 5.2 shows an intermediate state in the diag restructure. This is the state just prior to Line 15 in Listing 4.3. The arrows pointing down are child pointers, the arrows pointing up are parent pointers. A traversal at C will follow the right pointer to 4. Node 4 has no right child, so the traversal will proceed up the tree until it encounters C again. The traversal will re-process C and proceed to 4 again.

The diag restructure presents problems for traversals whether the traversal algorithm uses a stack or the parent pointers. Similar problems could be demonstrated with the zig restructure and swap. The root problem is that the relativistic algorithm presented in Chapter 4 assumed that readers would not use the parent pointers. Clearly the algorithm that uses parent pointers violates this assumption, but the stack algorithm also violates this assumption because popping the stack implicitly follows a parent pointer. For the traversals to be valid, either a new traversal algorithm must be found or the update algorithm must be modified to allow readers to

```
traverse(tree)
{
    node = leftmost(tree);

    while (node != NULL)
    {
        process(node);
        if (node.right != NULL) {
            node = leftmost(node.right);
        } else {
            parent = node.parent;
            while (parent != NULL &&
                   parent.right != NULL &&
                   node == parent.right)
            {
                node = parent;
                parent = node.parent;
            }

            node = parent;
        }
    }
}
```

Listing 5.2: Traversal algorithm that uses parent pointers instead of a stack.

traverse parent pointers. Section 5.2 presents both possibilities.

5.1.2 Duration

For the RBTree algorithm described in Chapter 4, the swap operation required a `wait-for-readers`. The length of `wait-for-readers` can be at least as long as the longest read-section. The length of a read-section is affected by the following three factors:

1. Time required to execute the read-section code
2. Time spent in blocking calls (I/O, lock acquisitions, etc.)
3. Scheduling decisions that cause the read-section to wait for other threads

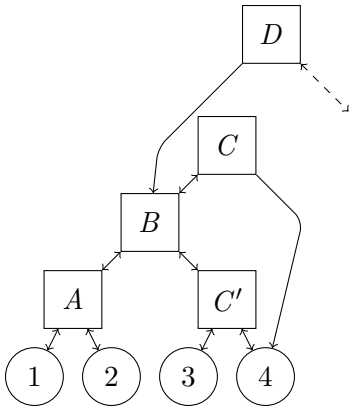


Figure 5.2: One of the intermediate states in a diag restructure. Down arrows are child pointers, up arrows are parent pointers. Traversals can get stuck in a loop including C and 4.

Scheduling decisions are outside the scope of this work, so we will assume a processor is always available. For simple lookups, the read-sections are of short duration for two reasons: Lookups are $\mathrm{O}\left(log(N)\right)$ and the read-section can be contained within the lookup that does not block.

For complete traversals, there is an interplay between the traversal code and user code. The traversal code finds the next node and then the user code performs an operation using that node. The user code has the potential to be blocking either for I/O or because it acquires a lock to perform some other operation based on the node value.

Traversals maintain references to nodes in the tree across calls to `next()`. The stack algorithm maintains these references on the stack, the parent algorithm requires a reference to a node be passed into `next` so the algorithm knows where to continue the traversal. Because the traversal maintains references to the tree across calls to `next()`, a read-section must encompass the entire traversal. Traversals, and therefore read-sections, can be quite long because a user mode action, which may include blocking calls, is invoked for each node in the tree.

The long duration of traversal read-sections can affect both update performance and memory usage. The swap algorithm required a `wait-for-readers` which can block for at least the duration of a read-section. This will significantly delay any updates that include swaps. Even if a series of updates could be constructed that never required swaps, there is another potential problem. Because traversals are of long duration, there is the potential for a large number of updates during the time of the traversal. Each of these updates could cause a node to be freed asynchronously. All of these asynchronous frees will be delayed until the traversal completes. For restricted memory machines, this could lead to memory exhaustion.

5.1.3 Linearizability

The previous chapter argued that the last `rp-read` in a lookup was the linearization point for that lookup. No such linearization point exists for complete traversals because the traversal does not depend on a single node—it depends on all the nodes in the tree. For locking solutions, the linearization point is the point when the lock is acquired. For the relativistic solution, the `start-read` is not a valid linearization point because the `start-read` does not determine the state of the tree that will be visible to the traversal. Updates can continue to occur following the `start-read`.

While it is true that it is difficult to assign a linearization point to a relativistic traversal, it is also true that having a linearizable traversal is less important than a linearizable lookup. If updates are independent, then a non-linearizable traversal may be acceptable. Consider the directory information a phone company maintains. Printing a phone book would constitute a traversal. Updates would consist of people signing up for service (inserts) or dropping service (deletes). If we consider the simple case of two updates, one person adds service another drops service, then these two updates are independent (provided they do not both involve the same phone number).

A phone book that included either, neither, or both updates could be considered correct regardless of the order of the two individuals in the customer service line. The duplicate phone number problem could be solved by deferring adding a dropped phone number into the available phone number pool until after all currently printing phone books have finished printing (e.g. use `defer-for-readers` to place dropped phone numbers into the pool).

Two of the solutions proposed in the next section are non-linearizable.

5.2 Three Traversal Algorithms

This section describes three approaches to solving the problems presented in the previous section. Table 5.1 gives a summary of the algorithms presented in this section.

Approach	Linearizable	Performance	Description
Combo	yes	O (N)	Use a reader-writer lock to protect traversals from updates. Lookups proceed relativistically independent of traversals or updates.
Simple RP	no	O $(Nlog(N))$	Have `next()` perform a lookup from the root at each step of the traversal.
Complex RP	no	O (N)	Use a more complex update algorithm that allows nonlinearizable relativistic O (N) traversals.

Table 5.1: Attributes of various approaches to traversals.

5.2.1 The Combo approach

This approach combines relativistic lookups with a more standard approach to traversals. The standard approach to a tree traversal is to treat the entire traversal as a single operation. A lock is acquired at the beginning of the traversal and held until the end of the traversal. The lock prevents any updates from happening during the traversal thus the structure of the tree remains unchanged for the duration of the traversal. To allow this type of traversal using the relativistic read and update algorithms described earlier, the mutex used for the write lock is replaced with a reader-writer lock. This approach yields three sets of critical section bounding primitives:

1. `start-read`/`end-read` bounds a relativistic read. These primitives are used for lookups. Relativistic reads do not exclude updates nor traversals.

2. `write-lock`/`write-unlock` bounds an update critical section. Updates do not exclude relativistic readers, but do exclude traversals.

3. `rw-lock`/`rw-unlock` bounds a traversal. Multiple `rw-locks` can be acquired at the same time. A traversal does not exclude other traversals nor relativistic readers, but does exclude updates.

Using these primitives, lookups can proceed at any time. Traversals and updates are mutually exclusive, so traversals will cause significant delays to updates. However, because traversals are mutually exclusive to updates, traversals are linearizable.

5.2.2 A Simple Relativistic Approach

Consider the following observations about tree traversals and the relativistic algorithms given in Chapter 4:

1. Traversals are O(N); Updates are O(log(N)) therefore traversals are expected to take much longer than updates.

2. Some updates require a `wait-for-readers` in the middle of the update. If the `wait-for-readers` must wait for traversals, the wait will be O(N). This will significantly delay updates.

3. The algorithms given in Chapter 4 assume readers do not access the parent pointers so parent pointers should not be used by relativistic traversals.

Given the above considerations, a relativistic traversal can be constructed using relativistic lookups. The `next()` primitive is passed the key of the previous node and returns the key and value of the node with the first key greater than the previous key. This allows the same relativistic read and update algorithms described in Chapter 4 to be used. The consequences are that a traversal will take O($N \log(N)$) time because each call to `next()` does a lookup starting at the root of the tree. However, `wait-for-readers` only delays for single calls to `next` rather than for the full traversal. This approach is possible because `next` does not need a reference to the current node, just the key of the current node. References into the tree do not need to be maintained across calls to `next`.

The tree traversed using this approach may not represent a tree that existed in a globally ordered time. In particular, it is possible that a great number of updates occurred during the time of the traversal. No guarantees can be made as to which of these updates were seen and which were not. The only guarantee is that `next()` will return the next node that was in the tree during the relativistic snapshot of time in which `next()` was called.

Using this approach, traversals will take longer; however, updates will not be significantly impeded by traversals.

5.2.3 A Complex Relativistic Approach

The previous section described an $O(N \log(N))$ approach to traversals that is compatible with concurrent updates. The $O(N \log(N))$ approach was used for two reasons: the update algorithms did not preserve the read-consistency of the `parent` pointers in the nodes, and it allowed `wait-for-readers` to only delay for single calls to `next()` rather than for the full traversal.

Section 4.2.2 mentioned two approaches to the zig restructure: one required copying a single node and a `wait-for-readers` the other required copying two nodes. Copying additional nodes can address both of the reasons for the $O(N \log(N))$ traversals. Copying additional nodes can preserve the read consistency of the parent pointers and can eliminate the need for `wait-for-readers` during an update. Preserving the read consistency of parent pointers means a relativistic traversal does not have to perform a lookup starting at the root for each call to `next`. Eliminating all `wait-for-readers` calls from updates means the read-section can span the entire traversal without delaying updates. The long read-sections will still delay memory reclamation and could thus lead to memory exhaustion on limited memory systems.

The user mode RCU library (available at `http://lttng.org/urcu`) has recently released code for an RBTree which copies additional nodes thereby preserving the read-consistency of the parent pointers and removing all `wait-for-readers` calls from the update path. The approach benchmarked here is similar to the approach implemented in the user mode RCU library. The algorithm is described in Appendix B.

Using the more complex approach, updates take longer than with the approach described in Chapter 4; however, traversals only take $O(N)$. As with the simple relativistic approach, traversals will not necessarily reflect the state of the tree as it existed in any globally ordered time.

5.3 Performance Trade-offs

Benchmarks were run to demonstrate the performance trade-offs of the three approaches. The following attributes were demonstrated:

1. The high traversal cost of the *Simple RP* approach (Figure 5.3)
2. The contended traversal penalty of the *COMBO* approach (Figures 5.4 and 5.5)
3. The tree size penalty of the *Simple RP* approach (Figure 5.6)

The data was collected on trees ranging in size from 100–1,000,000 nodes. Figure 5.6 shows how the algorithms scale across this range of tree sizes. Figures 5.3–5.5 are concerned with scaling across number of threads. These figures show data for a tree of size 10,000 nodes as representative of the other sizes.

The benchmarks were run on a four processor, quad-core Xeon machine (16 hardware threads) running Linux 2.6.32. The RP implementation was supplied by the Usermode RCU library mentioned above. Because traversals are so much slower than other operations, data was collected for ten seconds per sample rather than just one. The data reported are operations per ten seconds. Appendix A gives additional information on the data collection methodology.

Figure 5.3 shows traversal performance without a concurrent updater. All approaches scale, even *COMBO* which uses a reader-writer lock. For smaller tree sizes, *COMBO* does not scale because of contention for the lock variable. This figure makes clear the performance penalty of the $\mathrm{O}\left(N\log(N)\right)$ traversals in *Simple RP*.

Figure 5.4 shows update performance in the presence of concurrent traversals. The leftmost data point represents the uncontended update performance. The remainder of the points represent update performance in the presence of a varying number of readers performing traversals. The *Combo* approach has good uncontended perfor-

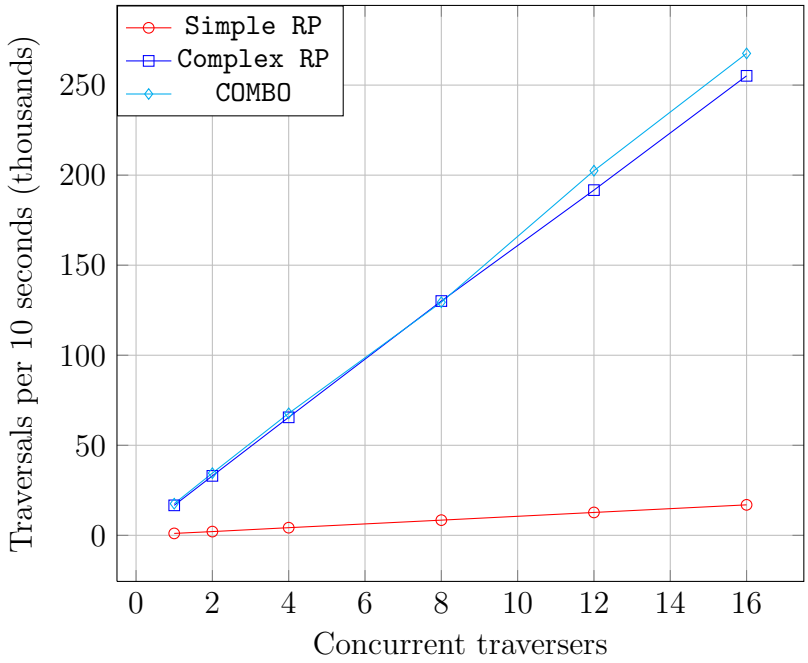


Figure 5.3: Traversal performance without a concurrent updater. Tree size was 10,000 nodes.

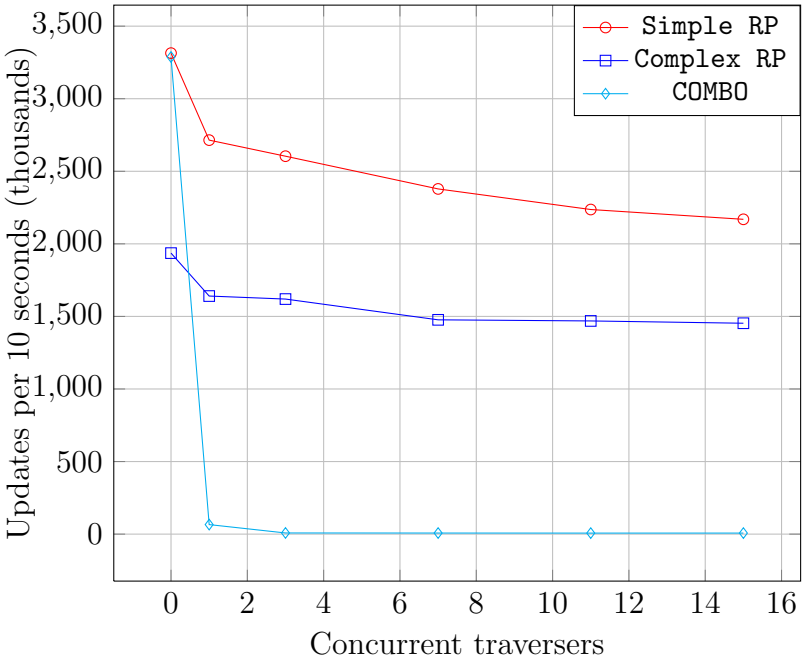


Figure 5.4: Update performance in the presence of concurrent readers performing traversals. Tree size was 10,000 nodes.

mance, but it degrades rapidly in the presence of traversals because each time a traverser wins the lock the update must wait for the traversal to complete. The *Simple RP* approach has best overall update performance, but this is at the expense of traversal performance as shown in Figure 5.5. Finally, the difference between *Complex RP* and *Simple RP* shows the cost incurred by *Complex RP* to allow O (N) traversals.

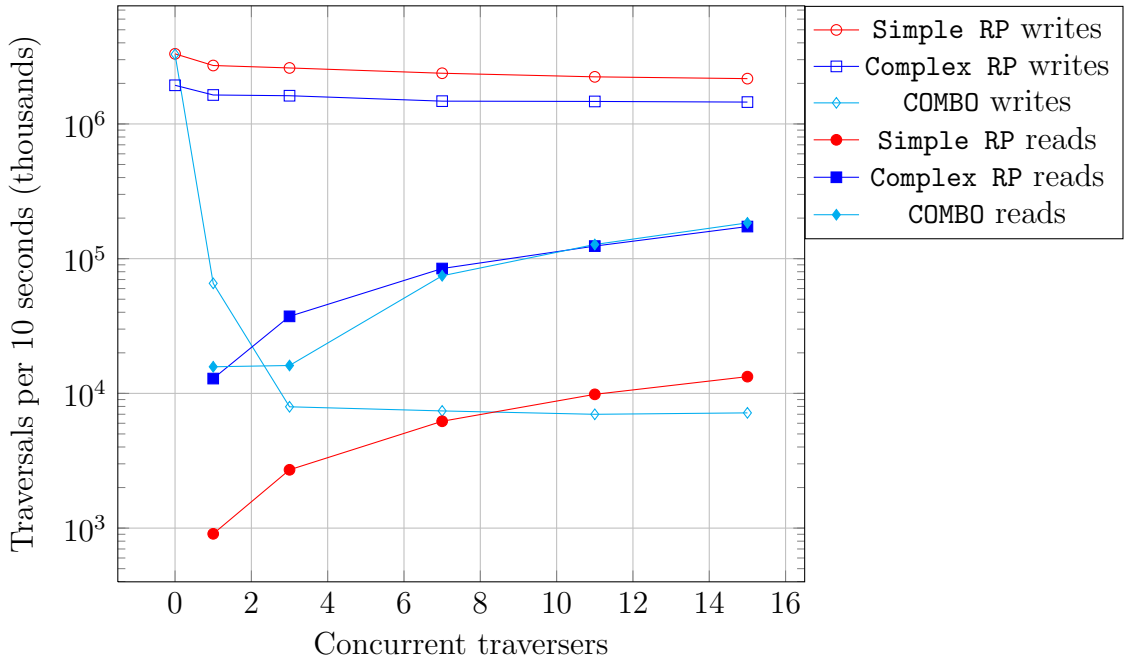


Figure 5.5: Update and traversal performance.

The reader-writer lock used by *COMBO* favors writers; however, because there is only a single update thread, overlapping writers cannot lead to read starvation. If a reader is waiting to acquire the lock, it can succeed before the write thread re-acquires the lock for the next update. In fact, if multiple readers are waiting, several can acquire the lock before the writer reacquires it. Figure 5.5 shows this behavior. In the presence of traversals, the update performance rapidly drops to roughly the traversal performance because following each update (or a small number of updates) a traverser wins the lock forcing the next update to wait for a complete traversal.

When there are enough traversals competing for the lock, several win the lock before the updater reacquires the lock, so the traversals scale even though the lock favors updates. The *Complex RP* and *Simple RP* mechanisms do not show this interplay between updates and traversals.

Figure 5.6 shows how the traversal performance scales with tree size. The data was collected with a single traverser and no updaters. The O (N) approaches show a drop in scalability between trees of size 10,000 and 100,000 nodes. The L2 cache is

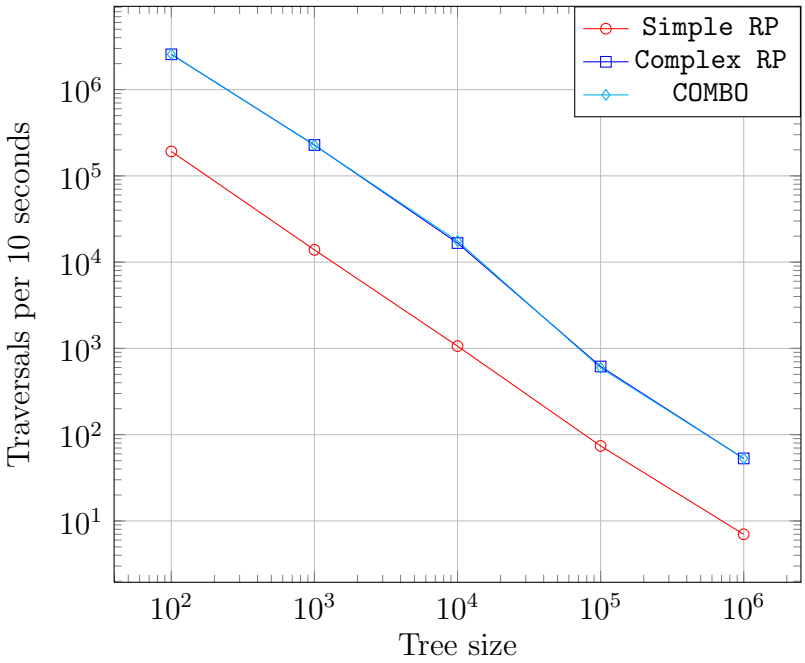


Figure 5.6: Traversal scalability. Note that the *COMBO* and *Complex RP* lines are on top of each other.

exhausted by trees between these two sizes. The access pattern in the O $(Nlog(N))$ approaches is more cache friendly so when the tree overflows the cache the *Simple RP* algorithm does not take as big of a hit.

5.4 Summary

This chapter has shown that read access patterns can have an impact on relativistic algorithms. Complete tree traversals present a different set of problems compared with individual lookups. With a complex read patterns such as in a complete traversal, there are several parameters that can be traded-off against each other. These are summarized below:

Linearizability Both of the *RP* solutions gave up linearizable traversals in exchange for concurrent updates. The *Combo* approach had linearizable traversals, but this meant the traversals and updates could not run concurrent with each other. The mutual exclusion between traversals and updates had a significant impact on update performance.

read peformance The *Simple RP* approach gave up traversal performance by using an $O(Nlog(N))$ traversal algorithm. This allowed concurrent reads (both lookups and traversals) and allowed better uncontended update performance than the *Complex RP* approach.

update complexity The approach used a much more complex update algorithm. The additional complexity means it is harder to prove this algorithm is correct. However, using the more complex algorithm allowed $O(N)$ traversals in the presence of concurrent updates.

Chapter 6

Disjoint Access Relativistic Writes Using Software Transactional Memory

Much of the work on relativistic programming has focused on a single writer at a time (enforced through mutual exclusion) and multiple readers. A truly scalable implementation should have both scalable reads and scalable writes. Some data structures can achieve a level of scalablity on the write side through partitioning, but as mentioned in Section 4.1, some data structures are difficult to partition. To solve this problem, relativistic readers can be combined with transactional writers. Using relativistic reads will allow the same high-performance scalable reads as with a standard RP implementation. Using transactional memory for the updates will allow automatic disjoint access parallelism on the write side.

This chapter is outlined as follows: Section 6.1 discusses the requirements for an RP compatible STM (RP-STM). Section 6.2 discusses SwissTM, an already available STM which supplies many of these requirements. Section 6.3 discusses the modifications made to SwissTM to make it RP compatible. This section also argues for the correctness of these modifications. Section 6.4 shows that the performance of RP-STM maintains the low overhead linearly scalable reads and also allows for scalable writes. Section 6.5 discusses a mechanism to extend the benefits of RP-STM by performing the lookup phase of an update relativistically. Section 6.6 summarizes the contributions of this chapter.

6.1 How to Make STM Compatible with RP

Chapter 3 discussed the ordering requirements for relativistic programming implementations. These were compared with the ordering requirements for reader-writer locks. The ordering requirements for STM systems are in some ways similar to those for fine grained reader-writer locks. Read-read concurrency is allowed, but not read-write nor write-write to the same data. Because readers are not allowed to see intermediate states, programmers only need to reason about correctness at transaction boundaries. If an STM is to allow concurrent relativistic reads, then intermediate states will be visible and the STM must be capable of maintaining the ordering requirements for a relativistic update.

The goal of the combined RP-STM system is that reads proceed completely outside the transactional system so they maintain the same performance as conventional RP reads. Writes proceed inside the transactional system so the STM can provide automatic disjoint access parallelism on the write side. The write algorithms need to be RP algorithms in order to support concurrent non-transactional reads. The RP update algorithm will keep the data always-valid. The STM system will provide isolation and the appearance of atomicity between updates. But because reads proceed outside the STM, the STM must maintain the ordering requirements of the RP algorithm.

Rather than developing an RP-STM from scratch, an existing STM implementation was modified to support the RP ordering requirements. A software transactional memory system that is compatible with relativistic readers must have the following properties (see Section 2.1.3 for a definition of the STM terms):

Weakly atomic A strongly atomic transactional memory system would include the relativistic reads as part of its atomicity guarantee thus requiring

significantly more overhead than relativistic reads. The RP-STM reads are supposed to execute completely outside the transactional system so they maintain the same performance of standard RP reads.

Invisible Uncommitted data Updates that eventually get rolled back must not be visible to readers. Read-only operations happen outside the transactional memory system so they will never get rolled back. Therefore, read-only operations must not see any writes that will not be committed. This requirement can be met by software transactional memory systems that use a re-do log rather than update-in-place.

Program Order The transactional memory system must honor the program order of writes to memory. Normal compiler optimizations and out-of-order execution units on modern processors are allowed, but a wholesale reordering as is done by many STMs is not allowed. In particular, all writes in program order prior to an `rp-publish` must be visible to readers prior to the `rp-publish`, and all `rp-publish` operations must be executed in program order. To implement this property, the STM must have a record of all STORE operations, the record must be kept in program order, and the STORE's performed with `rp-publish` must be tagged in the re-do log so that the required memory barrier can be inserted at commit time.

`wait-for-readers` The transactional memory system must honor `wait-for-readers` delays between specific memory writes. The `wait-for-readers` is required between the time that one write is visible to readers and the time a subsequent write is visible to readers. In an STM

system, the writes are made visible to readers at commit time, so the `wait-for-readers` must be performed at commit time. RP-STM records the request for a `wait-for-readers` in the re-do log. Commits will be delayed when this entry in the re-do log is encountered.

Any STM that preserves these properties will allow a relativistic update algorithm to be transactionalized so that readers can proceed relativistically, outside the transactional system.

6.2 SwissTM

RP-STM was derived from SwissTM [19, 18]. SwissTM was chosen as a starting point because it is a recent implementation that claims high performance and because it uses invisible reads and a re-do log rather than update-in-place. This section gives a brief explanation of how SwissTM works. Many of the details are left out. The intent here is to provide just enough detail to facilitate a description of the changes necessary to make SwissTM compatible with relativistic reads and to argue for the correctness of changes.

Table 6.1 lists the meta-data maintained by SwissTM. The `tx-count` is a global counter of committed transactions. It is used as a timestamp for values. The timestamp is used to validate read-values. The `read-log` is used for validating the read-set of a transaction. The `write-log` is used to store all the values written. At commit time, these values are written back to memory.

The `w-lock` and `r-lock` are used to control access to memory locations. When held, the `w-lock` prevents other transactions from writing the protected value. Reads are still allowed. The `r-lock` prevents other threads from reading the protected value. Rather than a pair of locks per word, a pair of locks is assigned to a block of

words. Multiple addresses hash to the same lock, but the reduced meta-data improves performance. The `w-lock` is acquired the first time a transaction performs a STORE to a location, which allows for early detection of write-write conflicts. The `r-lock` is not acquired until commit time. Read-only transactions do not acquire the `r-lock`, but they do delay reading any values for which the `r-lock` is held. Write transactions acquire the `r-lock` on their write-sets prior to updating any values. Doing so prevents readers from seeing partial updates because if they encounter a value for which the `r-lock` is held, they will delay reading it until after the transaction completes. At this point, the value will be marked with a newer timestamp, so the reader will re-validate all their values.

The major operations of SwissTM are listed below.

`Tx-Start` saves the current value of `tx-count`. Any values that were written after this time have the potential to cause an inconsistent set of values. When this is detected, all values in the read-set are re-validated.

`LOAD` Pseudo-code for `LOAD` is given in Listing 6.1. `LOAD` checks the value of `w-lock`. If the current transaction is the owner of the lock, then the value is returned from the `write-log` (i.e. the last value written by this transaction is returned). Otherwise, `LOAD` waits for the `r-lock` to be unlocked. If the `r-lock` value is greater than the saved `tx-count` value, then the read-set is validated, otherwise the value is added to the `read-log`.

`STORE` Pseudo-code for `STORE` is given in Listing 6.2. If the current transaction is the owner of the `w-lock`, `STORE` simply stores the new value in the `write-log`. Otherwise, `STORE` attempts to acquire the `w-lock`. If it succeeds, `STORE` checks the `r-lock` value to determine if the value has been written after the start of the transaction. If so, the read-set is validated. If `STORE` is

Data	Scope	Updated	Purpose
`tx-count`	Global	commit time	Used as a timestamp to speed validation
`read-log`	per transaction	on each LOAD	Record of memory locations read and the timestamp for which they were read
`write-log`	per transaction	on each STORE	Record of memory locations written and the values they contain
`w-lock`	per block of addresses	on each STORE	Early detection of write-write conflicts
`r-lock`	per block of addresses	commit time	Prevent reading inconsistent values. When an `r-lock` is unlocked, it contains the `tx-count` of the transaction that last updated the value thus the age of the value can be determined.
`my-tx-count`	per transaction	transaction start	Record of transaction start time.

Table 6.1: Meta-data maintained by SwissTM

unable to acquire the `w-lock`, it calls the contention manager to decide which of the competing transactions should be aborted.

`validate` The `validate` operation compares all of the values in the read-set with the current values. If any values do not match, the transaction is aborted.

`commit` The pseudo-code for `commit` is given in Listing 6.3. If the transaction was read-only (if the write-set is empty), commit returns immediately. The read-set has already been validated so there is no work to do. If the transaction was a write transaction, then `commit` begins by acquiring the `r-lock`

on all addresses in the write-set. This operation is guaranteed to succeed because the transaction already holds the `w-lock` on these addresses. `commit` then increments the global `tx-count` to indicate the epoch in which the new values will be written. `commit` performs a final validation of the read-set. Finally, `commit` goes through the write-log and for each entry, it updates the value at the address, releases the `w-lock` and sets `r-lock` to the current `tx-count` value marking the time when the value was written.

```
1  LOAD(addr)
   {
      if (w-lock(addr) owner == me) {
         return value from write-log
5     } else {
         while (r-lock(addr) is locked)
         {
             // wait
         }
10
         if (r-lock(addr) > my-tx-count) {
            if (!validate()) abort;
         }

15       save (addr, value) in read-log
         return value
      }
   }
```

Listing 6.1: Pseudo-code for LOAD operation

6.3 Modifications to SwissTM

SwissTM was chosen as a starting point because its features were closely compatible with the desired RP-STM. The primary goal in making changes to SwissTM was ease of implementation. This implementation was not optimized for performance,

```
STORE(addr,value)
{
   if (w-lock(addr) owner == me) {
      update write-log with value
   } else {
      while (w-lock(addr) is locked)
      {
          call contention manager
      }

      acquire w-lock(addr)

      if (r-lock(addr) > my-tx-count) {
         if (!validate()) abort;
      }

      save (addr, value) in write-log
   }
}
```

Listing 6.2: Pseudo-code for STORE operation

```
commit()
{
    if read-only return
    forall write-set
        acquire r-lock

    increment global tx-count

    if (!validate()) rollback and retry

    forall write-set {
        update memory
        release w-lock
        update r-lock with tx-count
    }
}
```

Listing 6.3: Pseudo-code for `commit` operation

but there was very little difference in performance between RP-STM and the original SwissTM (see Section 6.4.1 for details). The following changes were made to SwissTM:

1. Added a new log (called the rp-log) to store all updates in program order. The original log recorded only the most recent store to any particular address. The rp-log records each store—if there are multiple stores to the same address, they will all be in the rp-log. The original log is used for loading values from within a transaction and for validating a transaction prior to commit. The rp-log is used at commit time to perform the memory writes.

2. Changed the `STORE` primitive so that it wrote values to both the original write-log and to the rp-log (added a line following line 17 in Listing 6.2).

3. Added a primitive to add a `wait-for-readers` to transactions. This primitive adds an entry to the rp-log.

4. Added a primitive to force a memory barrier before particular writes as indicated by `rp-publish`. This primitive performs all the processing of a normal `STORE`, but adds a tag to the entry in the rp-log indicating that the store must be preceded by a memory barrier.

5. Added a primitive to add an `rp-free` to the transaction. This information is added to the rp-log.

6. Changed the commit code to make changes to memory based on the rp-log rather than the original log. The `forall` in Line 11 of Listing 6.3 was split into two loops. The first one goes through the rp-log and performs all the memory operations (including memory barriers, `wait-for-readers`'s, and `rp-free`'s). The second loop goes through the original log and frees the `w-lock`'s and updates the `r-lock`'s.

These changes required adding a new data structure and methods for updating that structure. The new structure was based on existing SwissTM structures (the `read-log` and `write-log`), so adding the new data structure was straightforward. Two existing routines were modified: one for performing `STORE` operations and one for performing the `commit`. In both cases the modifications were trivial. Finally, a new routine was added for performing the commit from the rp-log instead of the original write-log. This routine was just a loop which processed each element of the rp-log.

6.3.1 Correctness

This section presents a brief argument for the correctness of RP-STM. The focus is on two aspects: the impact of transactional updates on the relativistic reads, and the impact of the modifications on the integrity of SwissTM.

Transactional Impacts on Relativistic Reads

RP-STM makes changes to memory using the rp-log. This log includes all of the stores to memory as well as memory barriers and calls to `wait-for-readers`. When this log is replayed, the memory operations will occur in the same order, and with the same ordering primitives as if the non-transactional update algorithm had been run. The only difference is the amount of time between successive operations. In the RP-STM version, this gap is determined by how long it takes the commit procedure to process the next entry in the rp-log. In the non-STM version, this gap is determined by how long it takes the update algorithm to determine the next memory operation. But readers are oblivious to this gap. As pointed out in Chapter 3, provided the ordering primitives (`rp-publish` and `wait-for-readers`) are in the correct places, the timing or ordering of the other operations does not matter. RP-STM preserves the important ordering constraints, so relativistic readers can proceed with RP-STM the

same as they can with a standard RP implementation.

Another way to consider this argument is that with standard RP algorithms, the RP primitives are executed in line with the code. With RP-STM, the primitives are queued in the rp-log. At commit time, the rp-log is replayed such that all the primitives are executed in the same order as in the standard RP algorithm. The net effect of RP-STM is that the primitives are delayed in time. To a reader, this would appear the same as a scheduling decision whereby the standard RP updater was delayed in time. The delay does not affect the correctness of the update nor of the concurrent reads.

Integrity of the Transactional Memory System

The integrity of the transactional memory system is preserved for the following reasons:

1. The original SwissTM system used invisible reads. As a result, read-only transactions cannot invalidate a transaction that performs updates. Removing these read-only transactions from the transactional memory system has no impact on the validity of the transactions that perform updates.

2. Additional meta-data was added (the rp-log), but none of the original meta-data was modified. Isolation and the atomicity of updates are determined and guaranteed based on the original meta-data. None of the code that performs the validity checks was modified, neither was any of the code that enforces the atomicity guarantees. As a result, RP-STM will have the same conflict detection properties and atomicity guarantees as the original SwissTM.

3. The order and timing of updates to memory was changed. However, from a transactional point of view, a transaction will see either all or none of the

updates, so their order does not matter. Because the updates in RP-STM still take place between acquiring all the `r-lock`'s and releasing the locks, the isolation and atomicity guarantees are preserved.

4. The insertion of `wait-for-readers` within a commit will extend the duration of the commits. However, the transactional memory system must tolerate arbitrary delays during a commit because a committing transaction may get interrupted or rescheduled by the operating system.

5. The `wait-for-readers` does not block for any other transaction. It only blocks for relativistic readers that are outside the transactional system. As a result, these delays cannot lead to deadlock within the transactional system.

For these reasons, if the original transactional memory system is correct, the relativistic implementation is correct as well. No effort was made to prove the correctness of SwissTM, that was argued by its original authors.

6.4 Performance

The goals for RP-STM were to have reads with the same performance of standard RP reads while at the same time allowing writes to scale through disjoint access parallelism. To test whether this goal was met, a transactional version of the RBTree algorithm discussed in Chapter 4 was written. The same update algorithm was used, but memory accesses were changed to use the `STORE` and `LOAD` primitives so RP-STM could manage the operations. In addition, the transactional versions of `rp-publish`, `wait-for-readers`, and `rp-free` were used. The lookups were not transactionalized, so they occurred outside the RP-STM system.

For the benchmarks, a tree was created and initialized to a given size, then threads were started to perform operations on the tree. The number of operations performed

over the duration of one second was recorded. For each operation, the thread picked a random number, and then based on the number, the thread either performed a lookup, insert, or delete. The probability of insert was the same as the probability of delete so on average the tree remained the same size. The size of the tree for the benchmarks presented here was 64K nodes. Tests were also run on trees of 64 nodes. Comments in the text indicate where the smaller tree made a noticeable difference in the performance. The benchmarks were performed on a four processor, quad-core Xeon machine running Linux 2.6.32. Appendix A gives additional information on the data collection hardware and methodology.

The following algorithms were benchmarked:

rp This is the relativistic implementation described in Chapter 4.

ccavl The concurrent AVL implementation by Bronson et al. [10]. This is the best known concurrent tree algorithm, and it was compared against the STM versions to see if an algorithm customized to a particular data structure would out perform an STM implementation.

rpavl A relativistic AVL tree. The algorithm matches that described in Chapter 4 except that AVL balance properties were maintained instead of RBTree balance properties. This algorithm was compared against *rp* to see if there was a difference between an RBTree and an AVL Tree for the usage pattern in these benchmarks.

RP-STM The relativistic read, STM update combination described in this chapter.

SwissTM The SwissTM system described by Dragojevic et al. [19]. This was the original STM that RP-STM was based on.

SwissRP This is the modified SwissTM STM running a standard RBTree algorithm (not the RP algorithm). *SwissRP* was compared to *SwissTM* to determine the performance impacts of the changes made to *SwissTM*.

6.4.1 Preliminaries

Figure 6.1 compares the performance of *rp* and *rpavl*. While *rp* performed slightly better, they were close enough that comparing an RBTree algorithm against an AVL algorithm is fair.

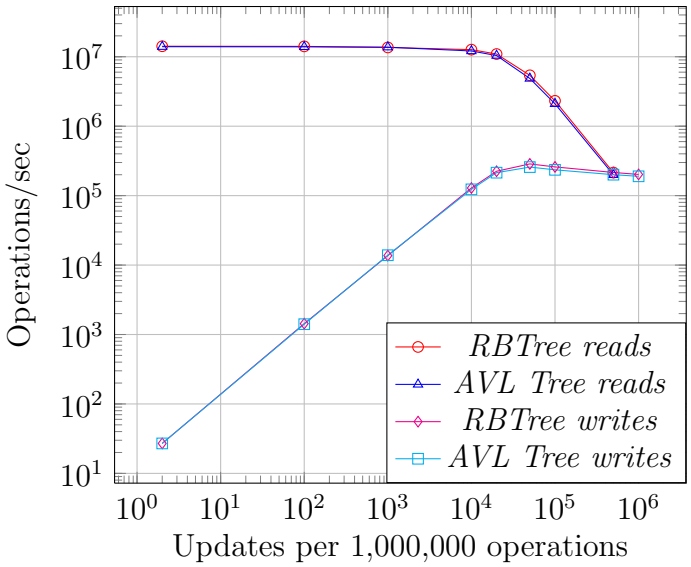


Figure 6.1: Performance of an RBTree compared with an AVL Tree across a wide range of update rates with 16 threads. The two performed sufficiently the same so that comparisons of RBTrees and AVL Trees are valid.

Figure 6.2 compares the performance of *SwissTM* and *SwissRP* (the modified STM but with non-RP algorithms). The performance of both STM's was virtually identical across a wide range of update rates. The changes made to the STM to make it compatible with relativistic reads did not have a significant impact on performance even though performance was not a consideration in making these changes. Because *SwissTM* and *SwissRP* have similar performance, any differences between *SwissTM*

and *RP-STM* must be due to the RP algorithms not due to the differences in the STM implementation.

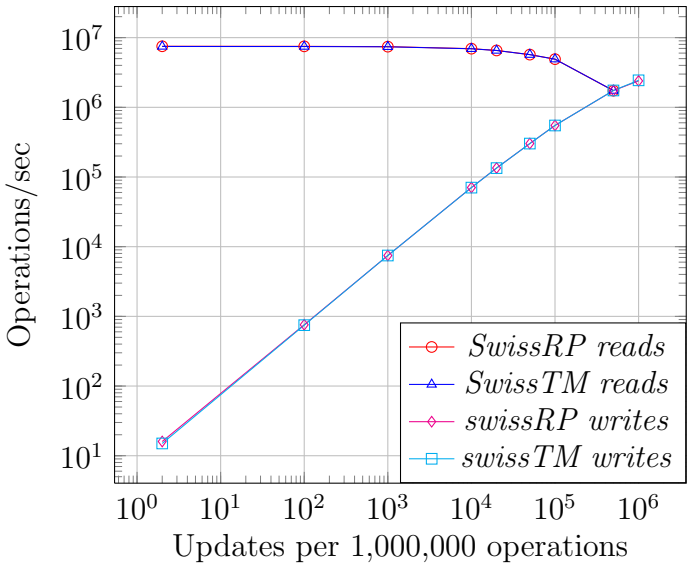


Figure 6.2: Performance of the original SwissTM algorithm and the modified version (labeled SwissRP) across a wide range of update rates with 16 threads. The modifications to make SwissTM compatible with RP reads did not significantly affect performance of the STM system.

6.4.2 Concurrent Reads

Figure 6.3 shows the performance of a read-only work load. As expected, the performance of *rp* and *RP-STM* was identical. The performance of *ccavl* was slightly better than *SwissTM*, but the additional performance does not justify the enormous complexity of *ccavl*.

6.4.3 Concurrent Updates

Figure 6.4 shows the performance of a write-only work load. As expected, the uncontended performance of *rp* was better than any other method because *rp* does not have to pay the STM overhead. However, *rp* degrades quickly in the presence of multiple

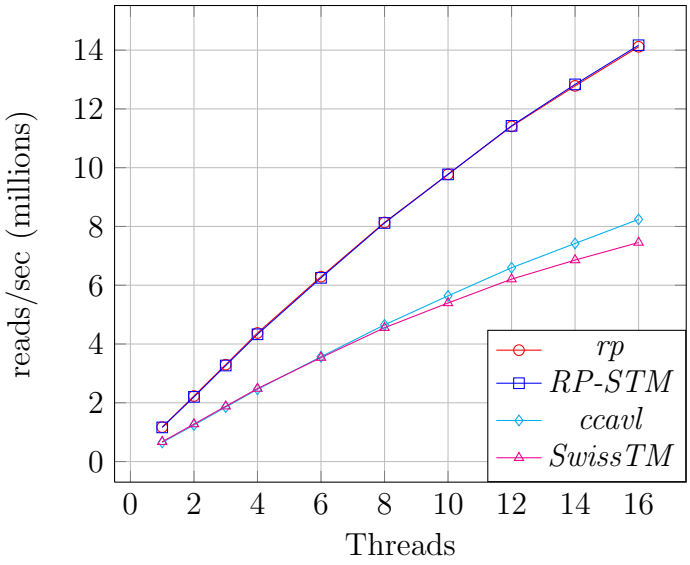


Figure 6.3: Read performance of various algorithms. The RP algorithms clearly out perform the others.

update threads because mutual exclusion is used between updaters.

Of the other algorithms, *SwissTM* out performed *RP-STM* because *RP-STM* used a more complicated update algorithm to allow reads to proceed relativistically. For low thread counts, *ccavl* was equivalent to *SwissTM*, but the performance of *ccavl* drops off significantly at higher thread counts.

For smaller trees, *ccavl* out performs the STM versions at low thread counts, but it again drops off at higher thread counts. For smaller trees, the performance of the STM versions levels off, but it does not drop at higher thread counts as *ccavl* does. The leveling off suggests that there is only so many ways a tree of a given size can be partitioned.

6.4.4 Variable Update Rate

Figure 6.5 shows the performance of a range of work loads from read-only to write-only. In each case, there were 16 threads performing work. The performance of both the *rp* reads and writes fell off significantly above about 5% updates because

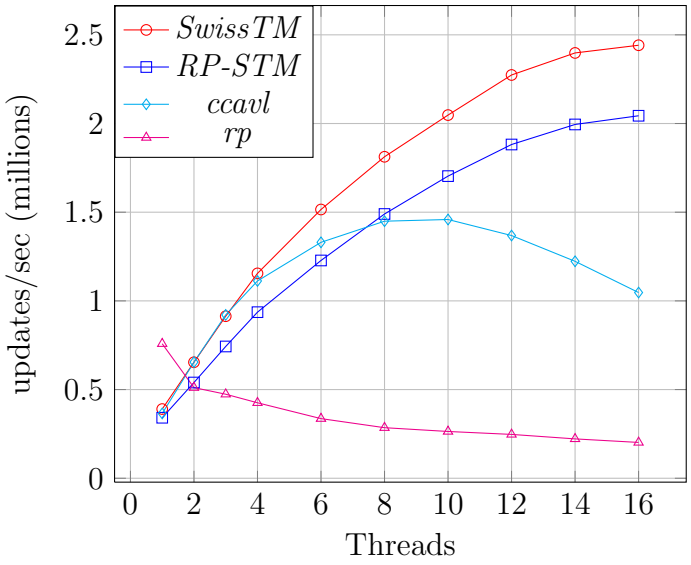


Figure 6.4: Update performance of various algorithms. The uncontended *rp* performance is better than that of the other algorithms, but *rp* drops off quickly due to mutual exclusion. The STM algorithms scale positively (though not linearly) out to 16 threads.

the mutual exclusion between writers caused the writes to serialize. Once a thread completed a write, it might perform several reads, but once it chose to do another write, it would have to wait until it acquired the lock thus preventing that thread from performing any work (reads or writes) for some period of time.

RP-STM showed the best performance across all but the highest update rates. At the highest update rates, *SwissTM* outperformed *RP-STM* because of the simpler update algorithm.

6.5 Discussion: Extending the Use of Relativistic Lookups

Updates for many data structures occur in two phases: a lookup phase and an update phase. Consider an RBTree insert. First, there is a lookup phase to determine the location for the new node, and then the node must be inserted and the tree rebalanced. It should be possible to perform a relativistic lookup followed by a transactional

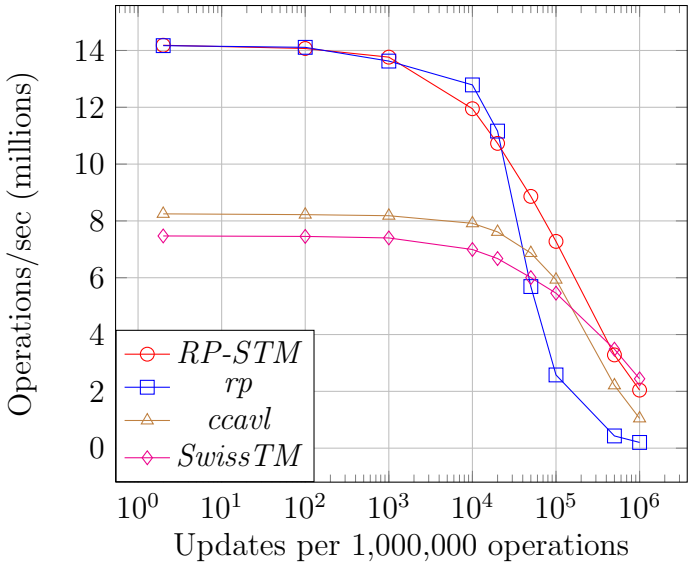


Figure 6.5: Combined performance across a range of update rates.

update. Performing the lookup relativistically has two benefits: It is faster and it reduces the read-set of the transaction, which should reduce conflicts thereby reducing rollbacks. Performing the lookup relativistically has one drawback: Following the lookup, there must be some validation within the transaction to confirm that the proper node is updated within the scope of the transaction.

Consider the tree shown in Figure 6.6. There are two simultaneous transactions. One transaction performs a zig restructure of nodes B, C, and E to rebalance the tree following some other operation, and the other transaction inserts H. The second transaction includes node E in its read-set. Node E is also in the write-set of the first transaction. Most STM's would report this as a conflict and one (or both) of the transactions would be rolled back. However, the insert of H does not really depend on E, only on G, the parent of H and the fact that G is the right child of its parent (if G is not the right child of its parent, then H may belong on the right branch of G's parent). These dependencies were not altered by the first transaction.

There are four considerations that determine if a relativistic lookup will be of

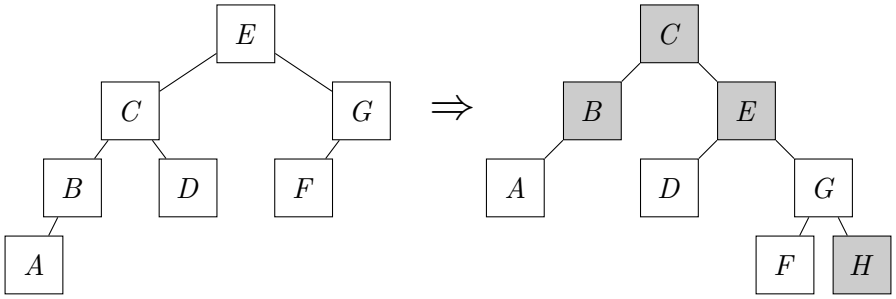


Figure 6.6: Subtree before and after the simultaneous insertion of node H and a zig restructure of nodes B, C, and E. The gray nodes are the nodes affected by the updates.

benefit.

1. Relative cost of a relativistic vs. transactional lookup
2. Probability that a relativistic lookup will avoid unnecessary rollbacks
3. Cost of a rollback
4. Cost of validation following a relativistic lookup

A preliminary analysis of the probability of a relativistic lookup avoiding an unnecessary rollback for an RBTree follows. The analysis shows that for RBTrees, relativistic lookups are not likely to avoid rollbacks. The analysis of the other three considerations for RBTrees and the analysis of other data structures is left as future work.

Using the following definitions:

LU_i the read set of the LookUp portion of the i^{th} operation

RS_i the Read Set of the non-lookup portion of the i^{th} operation (i.e. the read set of the operation after the node to be operated on is found)

WS_i : the Write Set of the i^{th} operation

Note that if the lookup phase proceeds relativistically, then LU_i is not included in the transaction.

For a relativistic lookup in operation i to prevent a rollback of operation k, the following must be true:

1. $LU_i \cap WS_k \neq \emptyset$
2. $(RS_i \cap WS_k = \emptyset) \wedge (WS_i \cap WS_k = \emptyset) \wedge (WS_i \cap RS_k = \emptyset)$

or, in English,

1. k wrote a location that i used to find the node to operate on
2. There are no conflicts in the transactional portions of i and k

To put this another way, the only way to have a conflict that matters is if $RS_i \cup WS_i$ (the nodes in the transactional portion of i) is below one of the nodes in WS_k. If this is true, then LU_i must include the node in WS_k that is a parent of $RS_i \cup WS_i$. If $(WS_k \cap (RS_i \cup WS_i)) \neq \emptyset$ then there is a conflict in the transactional portion of the operations, so an RP lookup will not prevent a conflict. See Figure 6.7 for a graphical representation of these relations.

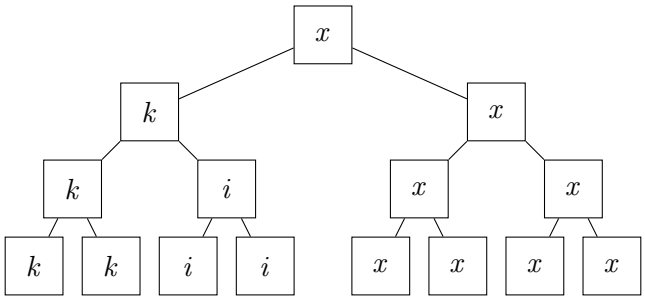


Figure 6.7: Tree depicting the read and write sets of two operations. The nodes in WS_k are marked k. The nodes in $RS_i \cup WS_i$ are marked i. All other nodes are marked x.

Consider the following: all inserts happen at leaves. Half of deletes happen at leaves (half the nodes in the tree are leaves - this is an attribute of all balanced

binary trees). The other half of deletes involve a swap with a leaf. Therefore, all updates involve a leaf.

Figure 6.7 illustrates a conflict that could be prevented with an RP lookup. The nodes marked k are in WS_k, the nodes marked i are in $RS_i \cup WS_i$, the nodes marked x are any other nodes in the tree. We assume that WS_k is the on left most branch of a tree (this is an arbitrary choice, but any other choice has an equivalent analysis). $RS_i \cup WS_i$ must be on a branch of some node in WS_k. Note that the size of $RS_i \cup WS_i$ is bounded by the size of WS_k as given in equation 6.1. This is because WS_k must extend down to a leaf and at most, $RS_i \cup WS_i$ can be a full subtree under the upper most node of WS_k.

$$|RS_i \cup WS_i| \leq 2^{|WS_k|-1} \tag{6.1}$$

I performed an empirical study and found that updates performed, on average, 3.5 `store` operations. This number remained constant independent of the size of the tree. Updating a single node may require multiple `store` operations—the parent pointer, color, and left and right pointers all may need to be updated. As a result, we can clearly conclude that, WS_k includes, on average, fewer than 4 nodes. Figure 6.8 shows a subtree with the four nodes in WS_k marked k. In order for there to be a conflict that was avoided by an RP lookup, WS_i must include one of the leaves marked i.

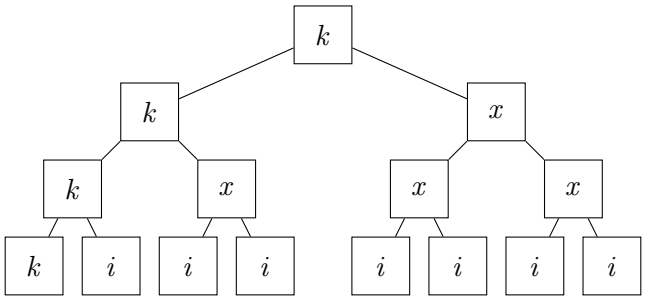


Figure 6.8: The nodes in WS_k are labeled k. If there is a conflict that was prevented by RP lookups, then WS_i must include one of the nodes labeled i.

If we assume that the leaf node involved in an update is randomly distributed among the leaves of the tree, then an upper bound on the probability of two concurrent transactions avoiding a conflict due to RP lookups is given by equation 6.2.

$$P(\textit{conflict avoided}) \leq \frac{2^{|WS_k|-1} - 1}{(\textit{tree size})/2} \tag{6.2}$$

This represents an upper bound because it assumes one `store` operation per node, it assumes WS_k does not include any siblings, it assumes a perfectly balanced tree, and because some of the leaves labeled i in Figure 6.8 would lead to transactions that would conflict even if RP lookups were used.

If WS_k includes 4 nodes (a liberal estimate based on the empirical study), then for trees of 64 nodes, $P(\textit{conflict avoided})$ is less than 7/32. As the size of the tree increases, the probability decreases.

The analysis presented above is preliminary. It needs to be tightened by determining the average number of nodes in WS_k instead of the number of `store` operations. It also needs to be extended to include the cost savings involved in doing RP lookups, the cost of validation following the RP lookup, and the cost of a rollback. The analysis should also include varying the number of concurrent transactions instead of considering just two.

While the analysis above is preliminary, it is sufficient to show that a relativistic lookup phase for RP-STM updates of RBTrees is not likely to reduce transactional conflicts for any but the smallest of trees. It is unknown what the performance impact of the relativistic lookup phase would be because no effort was made to measure the cost of the relativistic vs. a transactional lookup phase. Also, no effort was made to determine the cost of a validation phase. The analysis presented here was unique to binary trees. Other data structures, such as linked lists, might yield a different result.

These other considerations are left as future work.

6.6 Summary

This chapter showed that it is possible to combine relativistic programming with Software Transactional Memory in such a way that preserves the high performance, scalable reads of relativistic programming and also allows for automatic disjoint access parallel updates provided by the STM. While there was a slight update performance penalty for RP-STM, this is more than made up for by the high read performance. RP-STM provides a mechanism for scalable relativistic updates of data structures that do not easily partition, and the use of RP-STM is a good choice over a wide range of update rates.

Chapter 7

Summary of Contributions

This dissertation has contributed the following to the field of relativistic programming:

1. A new analysis of the ordering requirements of relativistic programming
2. Correctness criteria that can be applied to relativistic programs
3. Demonstrated that relativistic techniques can be applied to a complex data structure that requires multi-node updates
4. Demonstrated that `wait-for-readers` is useful for ordering the visibility of updates
5. Provided a mechanism whereby relativistic programming can be combined with software transactional memory to provide high performance, scalable reads, and automatic disjoint access parallel writes

Each of these are discussed below.

Chapter 3 presented a new analysis of the ordering requirements of relativistic programs. This analysis led to the following insights:

1. Reader-writer locks require symmetric two-way communication between readers and writers. Relativistic programs only require one-way communication. This

lower communication requirement is part of what allows relativistic programs to perform and scale better.

2. Relativistic programs, by default, do not require all readers to agree on the order of updates. `wait-for-readers` can be used to force all readers to agree on an order by forcing all readers that observed one state to finish before transitioning the data to a new state. If a single update requires multiple stores to memory, then `wait-for-readers` can be used to limit the number of intermediate states that are visible to a reader. If a `wait-for-readers` is placed between separate updates (as opposed to between the stores within a single update), then all readers will agree on the order of all updates. This behavior is called Totally Ordered Relativistic Programming (TORP).

3. The delays in TORP, where writes are delayed such that all readers will agree on the order of all writes, are no worse than the delays caused by reader-writer locking. The comparative delays were shown both analytically and experimentally.

4. If the "staleness" of data is measured based on the delay between the event that triggered a data change and when the data change became observable, then reader-writer locking increases the staleness of data over relativistic programs because reader-writer locking delays writes. Relativistic programming allows both earlier completion of writes and earlier access to data, meaning data is less stale.

Chapter 3 also proposed correctness criteria for relativistic programs. These correctness criteria were used throughout the dissertation when discussing the correctness of the various algorithms.

Relativistic programming techniques have long been applied to linked lists. Linked list updates tend to involve a single node at a time. Chapter 4 showed that relativistic programming can also be applied to data structures that require complex multi-node updates such as a red-black tree. The relativistic red-black tree algorithm made use of `wait-for-readers` to order the visibility of writes in order to abide by the relativistic correctness criteria. Chapter 5 extended this work further by looking at complete tree traversals. Several traversal solutions were presented that explored various points in the space defined by performance, degree of concurrency, algorithmic complexity, and linearizability.

Chapter 6 presented a Relativistic Software Transactional Memory. RP-STM can preserve the ordering requirements discussed in Chapter 3, so reads can proceed relativistically outside the transactional system. Doing so allows reads to have the same high performance and scalability of a regular relativistic implementation while the STM system can provide scalable writes through automatic disjoint access parallelism. An existing STM was used as the basis of the RP-STM implementation. Only minimal changes were required to the STM suggesting that compatibility with relativistic reads is a feature that could be added to other STMs. Also, the performance penalty for these changes was negligible, so adding this capability will not harm the performance of STMs, but it will allow a significant increase in read performance.

7.1 Future Work

There are a number of areas where the work presented in this dissertation can be extended. The work on ordering was useful for developing new relativistic algorithms (such as the relativistic RBTree) and for developing new mechanisms (such as RP-STM), but it is not sufficiently to formally prove properties of either the RP primitives or the algorithms that use these primitives. Such a formalism would be beneficial.

Relativistic programming supports read-read concurrency and read-write concurrency to the same data. RP-STM extends relativistic programming to support automatic disjoint access parallelism on the write side (write-write concurrency to disjoint sets of data). A complete relativistic solution should address a relativistic solution for both disjoint access parallelism on the write side as well as write-write parallelism to the same data.

One example of write-write parallelism to the same logical data would be to implement a First In First Out queue as an Earlier In Earlier Out queue. Each core could maintain a local queue so both enqueues and dequeues could be performed locally. The trick would be to bound the relaxation of order so there are limits on how much the dequeue order would vary from the dequeue order that would result in a totally ordered system. In other words, there must be bounds on the "Earlier" part of the definition of the queue.

A second example of relaxed consistency writes is the work by Unger et.al. [64] on inconsistency robustness. They proposed an OnLine Analytical Processing (OLAP) system that allowed conflicting writes. Rather than preventing the conflicts through locking, or detecting them and rolling back as in STM, they allow conflicting writes to publish potentially conflicting data. They use background threads to scan the data and find and fix inconsistencies. Their approach could probably benefit from relativistic techniques to place tighter bounds on the amount of allowable inconsistency.

Another area that could benefit from additional research is the area posed in Chapter 6 that used a relativistic lookup phase as part of an RP-STM update. The analysis presented in Chapter 6 showed that, for RBTrees, a relativistic lookup phase would not likely reduce the number transactional conflicts. However, even in the absence of fewer conflicts, the relativistic lookup would be faster. Additional research is needed to determine the method and cost for validating the selected node. Other data struc-

tures should also be analyzed because a different access pattern could significantly increase the probability of reduced conflicts.

7.2 Conclusion

Einstein is supposed to have said, "Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction." I believe relativistic programming is a step "in the opposite direction". Relativistic programming is not big and complex like STM systems, nor does it attempt to violently force a total order in a relaxed ordered world. Rather it is an elegant method by which ordering can be managed in such a way that efficient and scalable solutions are available to concurrent problems.

Bibliography

[1] Java platform standard ed. 6. [Online]. `http://download.oracle.com/javase/6/docs/api/java/util/TreeMap.html`.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, 1996.

[3] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, DAMP '09, pages 13–24, New York, NY, USA, 2008. ACM.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.

[5] O. Babaoglu, K. Marzullo, and F. Schneider. A formalization of priority inversion. Technical report, 1993.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.

[7] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 10.1007/BF00289509.

[8] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings*, pages 255–275, Portland, OR, June 1985. USENIX Association.

[9] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM.

[10] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 257–268, New York, NY, USA, 2010. ACM.

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM.

[12] Mark D. Campbell and Russ L. Holt. Lock-granularity analysis tools in SVR4/MP. *IEEE Software*, 10(2):66–70, 1993.

[13] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, September 2008.

[14] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[15] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[16] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, École Polytechnique de Montréal, December 2009. [Online]. Available: `http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf`.

[17] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2011.

[18] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 2010.

[19] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[20] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.

[21] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

[22] Rafael O. Fontao. A concurrent algorithm for avoiding deadlocks in multiprocess multiple resource systems. In *SOSP '71: Proceedings of the third ACM symposium on Operating systems principles*, pages 72–79, New York, NY, USA, 1971. ACM.

[23] Dennis J. Frailey. A practical approach to managing resources and avoiding deadlocks. *Communications of the ACM*, 16(5):323–329, 1973.

[24] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[25] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[26] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 245–257, New York, NY, USA, 1991. ACM.

[27] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[28] Justin E. Gottschlich, Jeremy G. Siek, Manish Vachharajani, Dwight Y. Winkler, and Daniel A. Connors. An efficient lock-aware transactional memory implementation. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 10–17, New York, NY, USA, 2009. ACM.

[29] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society.

[30] Sabine Hanke. The performance of concurrent red-black tree algorithms. Technical report, Albert-Ludwigs University at Freiburg, 1998.

[31] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15:745–770, November 1993.

[32] Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 280–280, New York, NY, USA, 2005. ACM.

[33] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[34] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[35] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distrib. Comput.*, 9:193–203, February 1996.

[36] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Language Systems*, 12(3):463–492, 1990.

[37] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230, Beverly Hills, CA, USA, March 1992.

[38] Jack Inman. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings*, pages 277–298, Portland, OR, June 1985. USENIX Association.

[39] K. H. (Kane) Kim. Basic program structures for avoiding priority inversions. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '03, pages 26–, Washington, DC, USA, 2003. IEEE Computer Society.

[40] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46:779–782, July 1997.

[41] Rob Landley. Red-black trees (rbtree) in Linux. *kernel.org documentation*, January 2007. [Online]. Available: `http://www.kernel.org/doc/Documentation/rbtree.txt`.

[42] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 157–168, New York, NY, USA, 2011. ACM.

[43] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 101–110, New York, NY, USA, 2009. ACM.

[44] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings*

of the 2008 37th International Conference on Parallel Processing, pages 67–74, Washington, DC, USA, 2008. IEEE Computer Society.

[45] Paul E. McKenney. Kernel korner: using RCU in the Linux 2.5 kernel. *Linux J.*, 2003(114):11, 2003.

[46] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf`.

[47] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. *SIGOPS Oper. Syst. Rev.*, 44:93–101, August 2010.

[48] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998. Available: `http://www.rdrop.com/users/paulmck/RCU/rclockpdcsproof.pdf` [Viewed December 3, 2007].

[49] Paul E. McKenney and Jonathan Walpole. What is rcu, really? *LWN.net*, 2007. [Online]. Available: `http://lwn.net/Articles/262464/`.

[50] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPoPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[51] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, University of Rochester, Rochester, NY, USA, 1995.

[52] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27:18–26, January 1993.

[53] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[55] P. J. Plauger. A better red-black tree. *C/C++ Users J.*, 17:10–19, July 1999.

[56] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. ACM.

[57] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[58] Bruce Schneier. Red-black trees. *Dr. Dobb's J.*, 17(4):42–46, 1992.

[59] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. ACM.

[60] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[61] Josh Triplett, Philip W. Howard, Paul E. McKenney, and Jonathan Walpole. Generalized construction of scalable concurrent data structures via relativistic programming. Technical Report 14, Portland State University, March 2011. `http://www.cs.pdx.edu/pdfs/tr1104.pdf`.

[62] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Scalable concurrent hash tables via relativistic programming. *SIGOPS Oper. Syst. Rev.*, 44:102–109, August 2010.

[63] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, Berkeley, CA, USA, 2011. USENIX Association.

[64] David Ungar, Doug Kimelman, and Sam Adams. Inconsistency robustness for scalability in interactive concurrent-update in-memory molap cubes.

[65] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 129–136, New York, NY, USA, 2006. ACM.

[66] Dieter Zöbel. The deadlock problem: a classifying bibliography. *SIGOPS Oper. Syst. Rev.*, 17:6–15, October 1983.

Appendix A

Benchmarking Methodology

This appendix discusses the data collection methodology used for the benchmarks presented throughout this dissertation. The data collection is discussed from a hardware, software, and statistical perspective.

A.1 Hardware

Two hardware platforms were used for data collection. Some specifics about each platform are given below.

The first platform was a Sun UltraSPARC T2 running SunOS 5.10. The UltraSPARC T2 has eight cores each supporting eight hardware threads for a total of 64 hardware threads. Each core shares an 8KB L1 data cache, and there is a single 4MB L2 cache shared by all eight cores.

One of the unique features of the UltraSPARC T2 is that atomic read-modify-write instructions bypass the L1 cache and execute directly out of the L2 cache. All atomic read-modify-write instructions pay the same communication cost independent of the amount of contention. If these instructions executed out of the L1 cache, there would be a different cost depending on whether the data was already in the L1 cache or elsewhere in the memory system. This aspect of the UltraSPARC T2's architecture affects the behavior of locking primitives that are based on atomic read-modify-write

instructions. On most architectures, there is a very noticeable drop in lock cost going from one to two threads because of the increased communication. This drop was not visible on the UltraSPARC T2.

The second platform was a four processor Intel Xeon (E7310) machine running Linux 2.6.32. Each processor has four cores so the machine supports 16 hardware threads. Each thread has its own core—there is no hyperthreading. Each core has a 2KB L1 data cache and each processor shares a 4MB L2 cache.

The Sun processor allows benchmarks to be run with more threads, however neither the User Mode RCU library nor SwissTM would build in the SunOS environment. Any benchmarks involving either the User Mode RCU library or SwissTM (and thus RP-STM) had to be run on the Intel platform even though it did not support as many threads.

A.2 Software

The general approach to collecting data was as follows:

1. Initialize a data structure to a given size
2. Start N worker threads which initialize and then wait to be told when to start performing operations
3. Set a global flag to start data collection
4. Delay for a fixed period (usually one second)
5. Reset the global flag to stop data collection
6. Each worker thread reports statistics on the operations performed

The worker threads could be arranged in one of three ways:

1. All threads are readers

2. One thread performs updates, the rest are readers

3. Each thread randomly chooses between a read and an update on each operation

There was a slight variation on the mechanism used to choose values to operate on based on which arrangement of threads was being used. For the first two, an array of size N (where N is the initial size of the data structure) was initialized along with the data structure. The array contained the keys that were also stored in the data structure. Reads selected a random number between $0..N-1$ and used that number as an index into the array. The value in that slot of the array was used as a key to perform a lookup in the data structure. This guaranteed that lookups would succeed in the read-only case.

The array was also used for the single writer case. The writer would select a random number between $0..N-1$ and use that number as an index into the array. The value in that slot of the array was used as a key to perform a delete from the data structure. A second random number was chosen and the writer would attempt to insert that value into the data structure. If the key was already in the data structure, the insert would fail. The writer would loop choosing new values until the insert succeeded. The just inserted value would be stored into the array in the same slot as the deleted value so the array continued to represent the values in the data structure. The range of key values was always $100N$ (where N was the size of the data structure) so the probability of collisions was kept low.

Because only a single thread was updating the array, no synchronization was needed for these updates.

The single writer case presented a window of opportunity for lookups to fail. If a reader chose the slot in array that was currently being processed by the updater, the

lookup may attempt to find the item that was just deleted by the writer. For a tree, the cost of looking for an item not in the tree is the same as the cost of looking for an item stored in a leaf, so these failed lookups were reported in the total number of read operations.

Most of the synchronization mechanisms discussed in this dissertation only allowed a single writer at a time due to mutual exclusion. As a result, only allowing a single thread to perform updates generated the maximum number of updates per second (because there was no contention for the write-lock). For these synchronization mechanisms, the important write-side data was the uncontended write performance and the effect of updates on read performance. The single writer case could generate both of these values. The uncontended write performance was generated when there was a single thread (the writer) and the impact on read performance was generated with a single writer thread generating the maximum write contention.

The random read/write arrangement was used when the synchronization method allowed writes to run in parallel. Under these conditions, the array could not be used to store the key values because there would be contention for the array and this contention could mask the data the benchmark was intended to collect. So instead of the array, the range of keys was set to $2N$ where N was the size of the data structure. For lookups, deletes, and inserts, a value in the range $0..2N-1$ would be selected and used as the key to perform the operation. The operations would succeed 50% of the time (there is a 50% probability the value would be in the data structure meaning the lookup or delete would succeed and there is a 50% probability the value was not in the data structure meaning the insert would succeed. Because the probability of deletes and inserts succeeding was the same, the data structure should stay approximately the same size. Also, this mechanism is self balancing. If more deletes succeeded than inserts, then the probability of future deletes succeeding would decrease and the

probability of future inserts succeeding would increase. The reverse is true of more inserts succeed than deletes.

A.3 Statistics

Each test setup was run sixteen times. The values reported in the figures throughout the dissertation is the average of the sixteen values. The standard deviation of the sixteen values was also computed. The figures do not contain error bars because there was very little variation from run to run. The error bars would not have been visible at the scale the figures were drawn in. There were a few exceptions to this rule. For example, in Figure 5.3 on Page 112 the standard deviation at high thread counts for the *Complex RP* and *COMBO* methods was such that the difference between the two at high thread counts could have been due to noise. However, repeated runs of 16 samples, and even runs of 32 samples, produced similar results, so the results were presented as-is without comment on sampling error.

Appendix B

Update Algorithm Compatible with $\mathrm{O}(N)$ Traversals

Chapter 4 described a relativistic algorithm for performing RBTree updates. The algorithm assumed that readers did not use the parent pointers in the nodes. The inability to use the parent pointers meant that complete tree traversals took $\mathrm{O}(N \log(N))$ instead of $\mathrm{O}(N)$. Chapter 5 mentioned, without giving details, a more complex RB-Tree update algorithm that allowed for $\mathrm{O}(N)$ relativistic traversals. This appendix sketches the main details of that algorithm.

Part of the correctness condition for the $\mathrm{O}(N)$ traversals is that the following relations must always hold. By "always hold" it is meant that there is never a reader visible state in which the relation does not hold.

1. `node->key == node->left->parent->key` if `node->left` is not NULL
2. `node->key == node->right->parent->key` if `node->right` is not NULL
3. `node->key == node->parent->left->key` if node is a left child
4. `node->key == node->parent->right->key` if node is a right child

Consider the delete of C shown in Figure B.1. To effect the delete, `B->right` must be changed to point to D and `D->parent` must be changed to point to B. Regardless which of these is changed first, the correctness condition will be violated until the second pointer is changed. The solution is to make copies of B and D as shown in

Figure B.2. The pointers between B' and D' can be changed before either of these nodes is made visible. When `E->left` is changed to point to B', the relations still hold even though in relation 3, if `node` is B, `node->parent->left` is B', not B. Similarly, prior to updating `A->parent`, if node is B', from relation 1, `node->left->parent` is B not B', but the relation still holds.

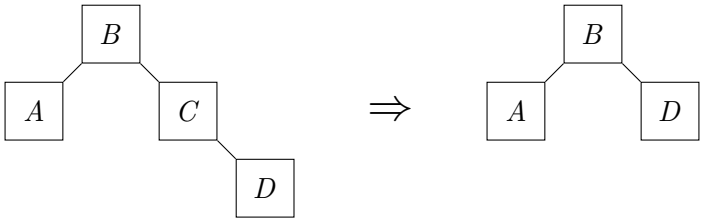


Figure B.1: If C is deleted, then `B->right` and `D->parent` must appear to both be updated atomically or else the correctness condition will be violated.

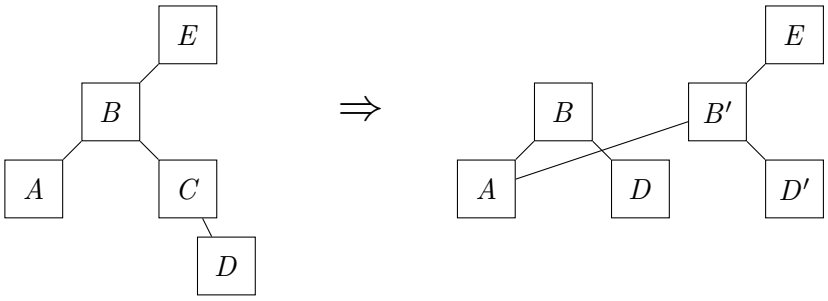


Figure B.2: Creating a copies preserves the correctness property because, for example, the correctness relation holds after `E->left` is updated to point to B'.

Figure B.3 shows a zig left rotation. In the original algorithm, only nodes A and C were copied. For the new algorithm, three additional nodes need to be copied. Comparing the before and after diagrams, any node that obtains a new parent or a new child must be copied. A similar analysis of the diag restructure means that three nodes need to be copied instead of just one.

Figure B.4 shows the delete of node B. This delete requires a swap with C. All of the nodes on the path from the deleted node to the swapped node are copied. In

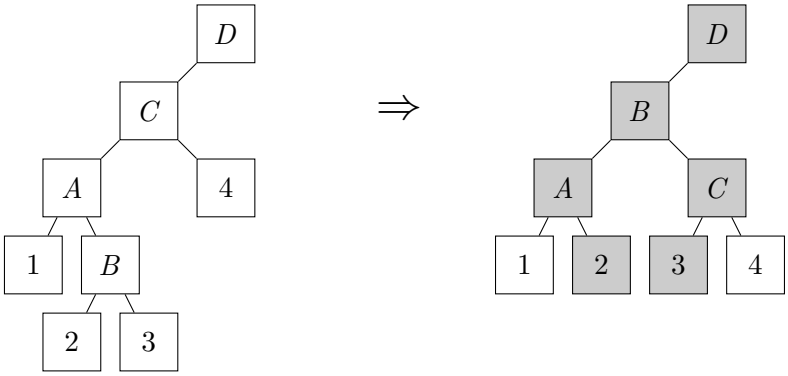


Figure B.3: Zig left restructure of nodes A, B, and C. The gray nodes are copies of the originals.

addition, the left child of the deleted node (A), the parent of the deleted node (H), and the right child of the swapped node (D) if it exists are also copied. For sufficiently large trees, and for a delete sufficiently high in the tree, the swap could be performed with two disjoint sets of nodes: one in the area near the deleted node and another in the area near the swapped node. This optimization was not taken.

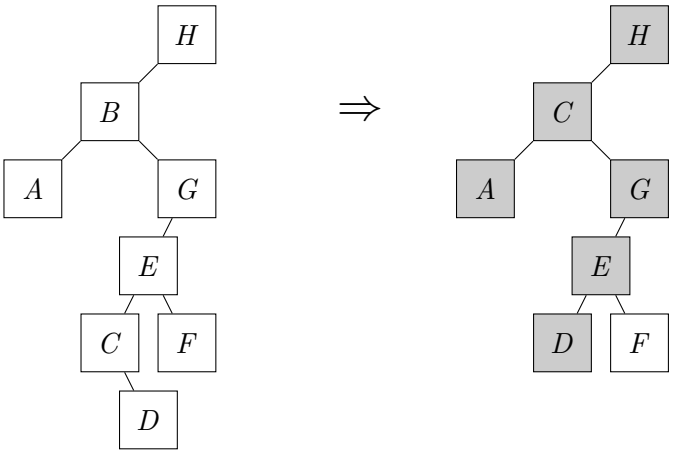


Figure B.4: Delete of B that requires a swap with C. The gray nodes are copies of the originals.