Concurrent Datastructure Design for Software Transactional Memory

Vrije Universiteit Amsterdam

Bachelor Project

Daniel Köves
`d.koves@student.vu.nl`

Supervisor
Prof. Wan Fokkink

Second reader
Dr. Thilo Kielmann

June 2021

Contents

List of Figures

Abstract

The following paper set out to investigate how transactional memory can be applied to concurrent datastructure design. Specifically, how it can provide a clean and accessible interface in replacement of locks and built on top of lock-free primitives for concurrent algorithms. Two of such datastructures are implemented, a transactional Red-Black Tree and a Skiplist, with underlying encounter-order and commit-time transactions taking care of concurrent insertions. Results show that word-based commit-time transactions cannot be successfully applied to Red-Black Trees without imposing an unnatural design on the datastructure, due to the fact that the commit-time transaction's write-list consist of (address, value) 2-tuples that are oblivious to chain of dependencies between the writes. For Skiplists, no such restrictions exist, with both encounter-order and commit-time transactions exhibiting optimal scaling properties. As opposed to findings of other papers like [1, 2], which promote the use of the commit-time mechanism for high-contention, this paper finds that due to the large amount of aborting transactions, which are expected in case of high-contention, and the more expensive commit-time API operations, the encounter-order locking mechanism outperforms commit-time locking on all metrics with a factor of two. Furthermore, it can be empirically concluded that the use of the transactional algorithm design is much less error-prone than its lock based counterpart, without further restrictions on its structure posed by lock-free primitives. While performance-wise, hand-crafted concurrent datastructures might prove to be more effictive, the transactional design can open up the process to a wider range of programmers by providing a straight-forward abstraction for concurrency.

Keywords— transactional memory, concurrent datastructures, parallelization

1. Introduction

"He who controls the spice controls the universe."
—Frank Herbert, *Dune*

1.1 Background

In the era of multi- and many-core systems, achieving parallelisation to a sufficiently high degree is of great importance. Speeding up sequential applications by parallelising them or consciously designing highly concurrent programs has been a goal in computing for some time now to fully utilise the available machine architectures and infrastructure. However, this attempt comes with a catch. In the 1960s, Gene Amdahl presented a theoretical upper bound[3] on the speedup of parallelised programs. It roughly states that even if a small amount of the program remains sequential (i.e. cannot be parallelised), the maximum achievable speedup will be rather limited. As an example, consider a program of which 75% can be fully parallelised and denote this fraction with p, and let $s = 1 - p$ be the remaining sequential work. The theoretical speedup $S(n)$, where n is the number of cores available, will than be no bigger than $S(n) \leq \lim_{n\to\infty} 1/(s + p/n) = 4$. With $p = 0.9$, the theoretical speedup bound is still only 10, even if we have an infinite number of processing cores available. In real-world applications, the implications of Amdahl's Law is not as daunting as it may first seem. Firstly, it assumes that the application operates on fixed problem size. However, today's massively concurrent machines allow computations on much larger datasets in the same amount of time. The argument that parallelisation is not just about speeding up programs but enabling them to deal with larger datasets is captured by Gustafson's Law[4]. Both Amdahl's Law and Gustafson's Law, however, fail to take communication into consideration. As systems get more and more complicated, so does the inherent complexity of communication between them; therefore, both laws paint a somewhat simplistic picture. Still, the main implication of both is that there are inherent limitations to parallelisation and that introducing n more computing cores does not immediately result in n-fold speedup.

Another problem involving concurrent programming is achieving synchronisation. Synchronisation refers to how separate concurrent *threads* manage and operate on shared data, i.e. ideally want maximum performance while ensuring that no concurrent operations overwrite each other or see inconsistent states. One of the more simple ways to achieve this is by using *locks*. Consider for example the simple *spinlock* with two operations: `lock()` and `unlock()`. Spinlocks (or locks in general) can be thought of as some global variable that threads acquire before executing operations on shared data. Those operations are said to reside in the *critical section*. If one thread wants to operate on a shared datastructure, it must first acquire the lock, perform the operations and then release the lock. In the meanwhile, other threads wishing to perform their work on the shared datastructure *spin*, until they can acquire the lock. One (of the many) problems with this simple way of achieving synchronisation, often referred to as *coarse-grained locking* is that it is slow and does not scale well to highly concurrent applications. When the computation runs on many threads, even if they want to modify completely disjoint parts of a potentially large datastructure, they must wait in line to perform their actions. Therefore, computation is still sequential, even though many computing cores are available. A possible alternative is to use *fine-grained locking* by associating certain parts of the datastructure with their own locks. That way, more concurrent threads can operate on different parts of the data. This can even be further enhanced and complicated by using *readers-writers* locks, i.e. when each locked part of the datastructure is associated with two locks; one must be acquired for reading the object while the other must be acquired when writing the object.

The above description about locks, by no means complete, is already quite complex and is still far from the whole picture. Mainly, it does not take into account how threads are scheduled by the operating system. Scheduling algorithms are *non-deterministic*, which, when applied to algorithms, generally means that given the same input, the algorithm might produce different outputs. Therefore, we cannot make any assumptions on how a thread is scheduled or when it is preempted or descheduled. This creates a problem for locking: *priority-inversion* happens when a lower-priority thread is preempted while holding onto a lock that a higher-priority thread needs. This means that the higher-priority thread needs to wait for the lower-priority one to be rescheduled and release the lock. Moreover, in datastructures where locks must be acquired one-after-another (also called *hand-over-hand lock-*

ing) to reach a certain part of the structure, *convoying* can occur, which means that if a thread further down the chain is delayed, all other threads behind it need to wait as well. *Deadlock* occurs when threads try to acquire the same locks in a different order and execution grinds to a halt.

A possible way to move away from locks is the use of *lock-free* primitives that are also detailed in Section 2.2. Hardware operations like *compare-and-swap* work by only allowing operations on shared data to succeed if the current value that a thread reads matches the actual value of the object (i.e. no other thread modified it in the meanwhile). These primitives, however, still have limitations and problems. Firstly, they usually operate on a single word in memory; therefore, only one location can be modified *atomically* (i.e. inseparably). Moreover, by this limitation, more complex lock-free algorithms tend to have a very unnatural structure.

What might be a solution to the problems described above? On the one hand, locks are performant; however, there are inherent problems with them. Apart from priority-inversion, convoying, and deadlocks, generally speaking, programming with locks is hard as complex situations can arise that are difficult to debug. Lock-free primitives propose an alternative; however, more complex algorithms tend to have an unnatural structure to them due to the limitations of the primitives.

A potential solution is *transactional memory*, a concept introduced by Herlihy and Moss in 1993[5]. Transactions, which are already an established phenomenon in databases, are a set of instructions operating on some shared data which have the properties that they are *serialisable* and *atomic*. Serializability means that transactions appear to execute one after another and never seem to interleave. Atomicity indicates that transactions make speculative changes to memory which they only make atomically visible by *committing* if no inconsistencies are found (i.e. no other transaction modified the shared data that the transaction accessed in the meanwhile). Suppose there is a conflict, the transaction aborts and can retry executing its operations. This provides a nice abstraction away from the "dirty details" of concurrent algorithms, as it is the transactional runtime system that takes care of the hard work. Wrapping instructions in transactions that modify a shared datastructure and retrying them until they are able to commit also provides a sufficiently straightforward interface, without the need to worry about locking, priority-inversion or unnaturally structured complex algorithms.

1.2 Goals

The goals of the paper are to present the history of transactional memory and, specifically, how it can be applied to concurrent datastructure design. Designing concurrent datastructures usually involve hand-crafting them to certain use cases, and the process involved is very demanding - often best left to experts. Using the transactional approach, however, the paper would like to show that designing transactional datastructures is not much different from designing sequential ones. In cases when the application performance isn't critical, the transactional interface can offer a very accessible approach to dealing with concurrency. The paper proposes two of such transactional datastructures that will be detailed in Section 4.2 and 4.3. Finally, an experiment is described in Chapter 5 to investigate the performance of certain transactional memory implementations underlying the datastructures and answer the research questions outlined below.

1.3 Research Questions

In order to realise the goals of the paper, the following research questions are explored related to transactional datastructure design:

1. How does the locking scheme of lock-based STM implementations affect the insertion performance of concurrent red-black trees and skiplists?
2. How well suited is transactional programming for concurrent datastructures in terms of ease-of-design?

Research Question 1 is explored thoroughly in Chapter 5, where the proposed STM implementations are tested on two transactional datastructures: a Red-Black Tree and Skiplist.

Research Question 2 is a follow-up and a minor one, where based on the results described in Section 5.2 further conclusions on the ease of transactional design can be drawn.

1.4 Organisation

The rest of the paper is organised as follows: Chapter 2 introduces common concepts and terminology regarding progress conditions and synchronisation primitives. Chapter 3 gives an overview of Transactional Memory both in hardware and software by presenting a handful of papers and their approaches. Chapter 4 details the implementations proposed by the paper, including two STM versions: one using encounter-order locking, the other using commit-time locking. Moreover, two transactional datastructures, a red-black tree and a skiplist, are described. Chapter 5 describes the evaluation of the implementations and presents and discusses the results. Finally, Chapter 6 gives the concluding remarks of the paper with possible future directions to explore.

2. Terminology

This section aims to give an overview of the terminology used throughout the paper. Firstly, progress conditions are detailed, i.e. different implementation strategies concerning concurrent methods of an object. Then, the idea behind the two most common lock-free hardware synchronisation primitives is described.

2.1 Progress Conditions

Progress conditions describe different ways an object's method implementation can act on concurrent pending invocations. A method implementation is said to be **non-blocking** if threads are able to make progress even when one thread is delayed[6]. An implementation is **blocking** if delaying one thread can prevent others from making progress[6].

2.1.1 Non-blocking progress conditions

A method implementation is **wait-free** if a thread with a pending invocation to such a method keeps taking steps, it will finish in a finite number of steps. This guarantees that every thread makes progress if it takes steps[6]. In practice, this is often inefficient and wait-freedom can be relaxed in multiple ways.

One such way is to settle for **lock-free** methods which require that only some threads make progress that has a pending invocation to a lock-free method of an object. This guarantees that the whole system makes progress, even though some thread might be delayed.

Another way to relax the wait-free condition is to guarantee progress under certain assumptions on how the threads are scheduled. A method implementation is **obstruction-free** if it is guaranteed that a thread finishes in a finite number of steps after any point it executes *in isolation*, i.e. in a period when no other threads take steps.

2.1.2 Blocking progress conditions

When relaxing the non-blocking property, the following grouping can be made. A method implementation is **starvation-free** if a thread is guaranteed to complete in a finite number of steps when all other threads with pending invocations are making progress.

Finally, a method implementation is **deadlock-free**, whenever there is an invocation to that method and all threads with invocations are making progress, *some* will finish in a finite number of steps[6].

2.2 Hardware Synchronisation

To achieve synchronisation, most modern processor architectures provide one of the two following primitives: the *Compare-and-Swap* or *Load-link/Store-conditional* instructions.

2.2.1 Compare-and-Swap

Compare-and-Swap (or *CAS* for short), shown in Algorithm 2.2.1, is an instruction that takes three parameters: an address $\mathcal{A}$ in memory, and expected value $\mathcal{E}$ and an new value $\mathcal{V}$ for for that address. Compare-and-Swap executes *atomically* the following actions: in case address $\mathcal{A}$ contains $\mathcal{E}$, $\mathcal{A}$ is updated to $\mathcal{V}$ and true is returned. Else, nothing is changed and false is returned.

Algorithm 1 *boolean atomically* CAS($\mathcal{A}, \mathcal{E}, \mathcal{V}$)

```
1: if $*\mathcal{A} = \mathcal{E}$ then
2:     $*\mathcal{A} \leftarrow \mathcal{V}$
3: else
4:     return false
5: end if
6: return true
```

An interesting inherent problem of this approach, known as the ABA problem, occurs when two concurrently executing threads try to modify the same location. Consider the scenario when thread T_1 reads location $\mathcal{L}$ and sees that its current value is A. Thread T_1 is preempted and thread T_2 is allowed to execute. Thread T_2 reads location $\mathcal{L}$, modifies its value to B, and then back to A after doing some work. When thread T_1 is scheduled back, it sees that the value of location $\mathcal{L}$ has not changed and continues executing. The execution scenario can prove to be incorrect, since thread T_1 is unaware of the undetected change of values to B (and potentially other side effects of that action) at location $\mathcal{L}$.

2.2.2 Load-link/Store-conditional

Another way to achieve synchronisation, which does not pose ABA-like issues, is with a pair of instructions Load-link `LL` and Store-conditional `SC`. The `LL` instruction loads the value stored in address $\mathcal{A}$. Subsequent `SC` call to that address with a new value $\mathcal{V}$ succeeds only if the value of $\mathcal{A}$ has not changed since the corresponding `LL` call. The `SC` instruction fails if the value at $\mathcal{A}$ has changed since the `LL` call; therefore, also detecting the ABA problem.

3. Literature Review

This section gives an overview of the history of *Transactional Memory*, starting from Herlihy and Moss' 1993 paper, in which the term was coined[5]. Following it, *Software Transactional Memory* is presented through multiple proposed versions like the original non-blocking implementation of Shavit and Touitou[7], Herlihy's DSTM[8] and Fraser's OSTM[9, 10]. Finally, the proposed *blocking* implementations are detailed with attention to Ennals' STM[11], but also the *Transactional Locking I & II* algorithms proposed by Shavit et al.[1, 2].

3.1 Hardware Transactional Memory

Transactional Memory was introduced to the distributed computing world by Herlihy and Moss in 1993[5]. In their paper, the authors detailed a new multi-processor architecture providing lock-free synchronization, which would enable programmers to specify read-modify-write operations on several words in memory, via transactions. It is implemented as an extension to already existing cache-coherence protocols, and require specialised hardware support in terms of instructions such as `LT` "Load-transactional", `ST` "Store-transactional", etc[5]. Their proof of concept describes transactions as a *serializable* and *atomic* finite sequence of instructions. Serializability refers to the fact that transactions appear to happen one after another and they never appear to interleave. Atomicity indicates that transactions make tentative changes to shared memory and either *commit*, by making its changes atomically visible to other transactions or *abort*, by discarding its changes[5].

3.2 Non-blocking Software Transactional Memory

3.2.1 STM of Shavit et al.

In 1996, Shavit and Touitou proposed an alternative transactional memory design which is implemented entirely in software, called *Software Transactional Memory*[7] (or *STM* for short). Their design builds on top of that of Herlihy and Moss, however, it can also be implemented on existing machines with *Load-link/Store-conditional* operations[7]. The paper focuses on implementing *static transactions*, where the data set is known in advance, and therefore, each transaction can be viewed as an atomic procedure storing the given new values[7]. Their STM implementation is also *non-blocking*, which means that a thread is guaranteed to make progress even when other threads are suspended.

Implementation wise, the original STM approach was as follow. Two shared datastructures coordinate the processes and their transactions: a vector `Memory` of size M which contains every transactional memory block, and a vector `Ownership` of the same size pointing to *records* that determine which transaction owns a certain block of memory. Records are shared structures that store information about the current transaction its corresponding process started.

When executing a transaction, a process's record is initialised and status set to *stable*. Afterwards, ownership for all involved locations of the dataset needs to be acquired in some increasing order. If that succeeds, the transaction writes the old values to the locations' records, calculates the new values and writes them to the memory locations, and finally, sets its status to *success*. If the ownerships cannot all be acquired, the transaction sets its status to *failure, failadd* where *failadd* is the address for which the ownership couldn't be acquired[7]. In case of a failed transaction, the algorithm utilises a *cooperative method* to help the other transaction holding ownership of the location needed for the failed transaction. This is done as follows: if the failed transaction is not already a *helping transaction*, it first releases all ownerships that it previously acquired and helps the transaction that holds ownership to the location which failed by becoming a helper for that transaction[7].

3.2.2 Herlihy's DSTM

Whereas the original STM algorithm of Shavit and Touitou was *static*, where the transactions and memory usage is known in advance[7], the first *dynamic* Software Transactional Memory implementation was proposed by Herlihy et al. in 2003, known as *DSTM*[8]. The algorithm allows dynamic creation of transactions and transactional objects, moreover, transactions can determine their course of action based on previously read (transactional) values, which makes it suitable to operate on dynamic datastructures[8]. It inherits the *non-blocking* and *lock-free* properties, i.e. a suspended thread will not prevent others from advancing.

The algorithm described in the paper is originally implemented in `Java` and its workings can be described as follows. The *DSTM* algorithm operates on *transactional objects* which are accessed by transactions. These transactional objects are a wrapper around conventional objects, which can be modified by first *opening* the transactional objects and modifying its contents[8]. These changes are only *speculative*, i.e. they have no observable effect until the transaction commits. The interface the authors provided consist of the following: firstly, the `TMThread` class represents transactional threads, which is implemented as a wrapper around `Java`'s thread class. The thread class holds a status field which can either be `ACTIVE`, `COMMITTED` or `ABORTED`. Secondly, the `TMObject` class represents transactional objects accessed by the threads, which can be opened in `READ` or `WRITE` modes. Each transactional type must have the `TMCloneable` interface implemented which provides a `clone()` method used for speculative modification[8].

After a transaction begins, it must call `open()` with an appropriate mode on the transactional objects it wishes the change. This creates a working copy of the object (using the before-mentioned `clone()` method) which the transaction can safely modify, called the *version* of the object. Each thread maintains a read-table of opened objects. The version of an object is determined by the status of the transaction that last opened it in `WRITE` mode. The `TMObject` class holds a reference to a `Locator` object, which in turn holds two versions of each object and a reference to the transaction that last opened it in `WRITE` mode. If that transaction commits, the *new* version of the object becomes valid, else the *old* version. The level of indirection by introducing the `Locator` object in between an object's metadata and its contents is necessary to be able to atomically change the three field of the `Locator`, and therefore, the `TMObject` itself. This is performed by swinging the reference in the `TMObject` class to a new `Locator`[8]. After each open call, the transaction is *validated*, to ensure that the accessed object's *version* is not outdated yet, i.e. another transaction hasn't committed and modified the object in the meanwhile. This is performed by checking each entry in the read-only table and confirming that the local version of an object is still the latest committed version of the object and that the transaction is still active and has not been aborted[8]. Committing a transaction involves validating its read-table and using a *CAS* operation to change its status from `ACTIVE` to `COMMITTED`[8].

3.2.3 Fraser's OSTM

The STM implementation proposed by Fraser in [9] and summarised by Fraser and Harris in [10] titled *OSTM* for *object-based* STM, shares some of the key elements of Herlihy et al.'s *DSTM*[8].

The algorithm is both *obstruction-free* and operates on *objects* as the unit of concurrency, which contain *references* that can be opened for transactional access[9]. Each transaction maintains *read-only* and *read-write* linked-lists of *object handles* which point to *object-headers*. These headers are used to point to the most recent version of an object that is accessed by a transaction. When an object of opened from write access, a *shadow copy* of the object is created. The shadow copy is used to make tentative changes to an object, which will become visible to the application upon successful commit[9].

Transactional commits contain two phases: the *acquire* and *release* phases. In the acquire phase, each opened object's header must be acquired in some efficient order, by replacing the object handles with the transactions *descriptor*. If a header is already acquired by another transaction, that transaction is aided to finish[9]. After the acquire phases, based on its outcome, the transaction's status is changed atomically to either successful or aborted. On success, in the *release* phase, each object has its header swinged to its shadow copy by a *Compare-and-Swap* operation[9].

Three issues are mentioned by the authors with the current design. Firstly, in order to avoid a performance bottleneck when opening objects for read access on datastructures where there is a single entry point, the algorithm only acquires headers for objects in the read-write list which is followed by a *read* phase. In the read phase, the algorithm checks whether the version of an object in its read-only list matches that of the object when it was first opened. In case they do, the algorithm can commit, else it must abort[9].

Secondly, in some cases, an object may be updated after the read phase, but before the transaction status is updated and its changes are visible. To avoid this, transactions report a new status *read-checking* in their read phase and ensuring that the transaction is such a phase atomically commit or abort. This is done by helping a transaction in its read phase reach its *decision point*, i.e. when its accessed objects modifications become visible to others, or the transaction aborts[9].

Thirdly, in the case when a cycle of transmissions wishing to read an object owned by the next transaction, to avoid deadlock, the transactions can abort others. To avoid livelock, not every transaction should be aborted, therefore, only the transaction with the lowest descriptor address can make progress. This ensures that the cycles are broken and the implementation remains *obstruction-free*[9].

3.3 Blocking Software Transactional Memory

3.3.1 Ennals' STM

While non-blocking STM implementations were gaining momentum, in 2006, Robert Ennals suggested that the non-blocking property is actually detrimental to performance. He showed that by not guaranteeing the transactions to be obstruction-free, the performance of STMs can be increased significantly[11]. The arguments for this proposal stems from the fact that while the non-blocking property is essential in distributed computing, it is not much so in non-distributed STM[11]. Ennals argues that porting multi-threaded programs to STM will be done by converting atomic blocks to transactions. In this case, a transaction blocking a lower or same priority one is acceptable since the same behaviour would be observed in the non-parallelised version of the program[11].

The paper then argues for the unnecessity of obstruction-freedom, which boils down to three claims. The first claim is that long-running transactions cannot block other transactions. This is refuted by the fact that obstruction-freedom only guarantees progress for nonconflicting transactions, and therefore, long-running transactions either have to be able to block conflicting transactions or have to be preempted[11]. The second argument for obstruction-freedom tells that the OS might halt if a task is switched out that holds critical resources. The authors identify that these context-switches are rare and not of great importance since a proper runtime system should identify and adapt the number of tasks to the number of cores available. Therefore, temporary switches that cause the system to block are acceptable and have an overall slight effect[11]. The third and last claim proposed by those in favour of obstruction-freedom gives that the OS might halt if a thread fails. This is rebutted by the fact that, either in the case of a software failure or in the case of a hardware failure, non-blocking STM would fail as well[11].

The authors point out two optimizations that cannot be applied when obstruction-freedom is a goal:

1. Storing object metadata inline, making extra cache misses unlikely
2. Bounding the number of active cores by the number of active transactions, to avoid unnecessary conflicts

The first optimization is necessary since previously described non-blocking STM implementations[9, 8] require multiple lookups to find the current version of the object from the metadata, which leads to a performance loss. Obstruction-freedom does not permit inline metadata, since two transactions $\mathcal{A}$ and $\mathcal{B}$ working on the same object can only safely execute when $\mathcal{A}$ blocks $\mathcal{B}$ until it has finished writing the object[11]. The rationale behind the second optimization is that a non-blocking STM implementation does not wait to start a new transaction when all cores of a machine already execute a transaction. This creates a bottleneck, as the number of concurrent transactions is directly proportional to the number of conflicts among them, which then decreases overall performance[11].

Implementation-wise, Ennals' STM includes two main points. Firstly, it uses *encounter-order locking* on objects that a transaction wishes to write. This entails holding onto locks until the transaction either commits or aborts[11]. Secondly, the authors utilise optimistic control over reads, which involves not locking objects when reading them, but recording the object's current *version number*. If that number remains the same throughout the execution of the transaction, or it only changes by the actions of the transaction itself, the transaction can safely commit, else it must abort.

3.3.2 TL

The claims of Ennals were endorsed by Dice and Shavit in a follow-up paper, in which they detail a new STM algorithm titled *Transactional Locking*[1], that outperforms the proposed blocking algorithm of Ennals[11]. In their paper, the authors argue that lock-based STM implementations outperformed their non-blocking counterparts on a variety of use-cases[1]. They point out that while the STM of Ennals[11] uses *encounter-order locking*, this mechanism only performs well on uncontended datastructures. The algorithm proposed by the paper[1] uses *commit-time locking*, which, according to the authors fits well into the memory model of languages like `C` and `C++`[1]. Moreover, as their results suggest, the approach taken by the *TL* algorithm scales better on contention ranges and even when *encounter-order locking* does perform better, it is small enough to be insignificant[1].

The transactional locking algorithm involves two modes: *encounter mode* and *commit mode*. In both modes, all transactional memory locations are associated with a *versioned-write-lock*, which uses a *Compare-and-Swap* operation to acquire it and a *store* operation to release it. A single bit of the lock indicates whether the lock is taken and the rest of the bits are reserved for the *version* of the location, which is incremented on every lock release.

In commit mode, all transactions maintain *read* and *write* sets. A transactional read first checks whether the address it wishes to access appear in the write-set using a *Bloom filter*[1], and if so, it loads the current value written to that address. If the write-set does not include the address at hand, the associated lock is fetched and its version is saved in the read-set. If the lock is taken, the transaction may either spin or abort[1]. Transactional stores are performed such that the address and value of the location are saved in the write-set. Periodically, the read-set is validated to ensure consistent states, and upon encountering inconsistent states, the transaction is aborted[1]. When committing, the locks are acquired for each store location (as given by the *commit-time locking* mechanism). The acquire step is performed with a single *CAS* operation, which acquires the lock and validates the current version of the location[1]. After this step, all locks associated with read locations are validated. If the versions matched, the transaction is considered to be *committed*. Otherwise, all locks are released and the transaction is aborted[1]. Finally, all locks are released associated with write locations by incrementing the corresponding version numbers and clearing the lock bits using a store operation[1].

Encounter mode operates similarly to the algorithm described by Ennals[11]. Here, read and write-sets are also maintained, however, locks are acquired for the write locations at the point when they are encountered, after which the address and value of the location are saved in the write-set. For transactional loads, the behaviour depends on the associated lock. If the lock is found to be unlocked or held by the transaction itself, the location is simply read. Otherwise, the transaction spins on the location[1]. Similarly to *commit mode*, the read-set is periodically validated to avoid observing inconsistent states, in if such inconsistency is found, the transaction is aborted[1]. When trying to commit, all locks are acquired corresponding to write loca-

[1]Bloom filters are probabilistic datastructures that permit testing for membership in $\mathcal{O}(1)$ time

tions, which involves a *CAS* operation that increments the version number as well. Following, the read-set is revalidated. If an inconsistency is found, the transaction is unrolled and may be retried. Otherwise, the transaction is considered to be *committed*. Finally, all locks associated with write locations are released by incrementing the corresponding version numbers and clearing the lock bits[1].

3.3.3 TL2

Later in the same year, Shavit et al. proposed a refinement on the *TL* algorithm, titled *Transactional Locking II* or *TL2*. The authors point out that two limitations of earlier STM implementations remain that prevent them from being deployed. Firstly, non-blocking STM implementations require *closed memory systems*, and their blocking counterparts either require the same, or need specialized allocators in terms of `malloc()` and `free()` implementations[2]. Secondly, in order to ensure that transactions only operate on safe data, specialised runtime environments are employed that could contain such irregularities[2].

The algorithm overcomes the potential problems with earlier non-blocking and blocking STM implementations in the following ways. Firstly, it does not require specialised `malloc()` or `free()` implementations, as they fit into the memory model of low-level languages by operating on an *open memory system*. This is achieved by using the associated lock of an object which ensures that a freed location cannot be written by a transaction[1]. Secondly, the algorithm only operates on consistent memory states, which removes the need for specialised runtime environments[2].

The *TL2* algorithm can be described as a variant of the *TL* algorithm of Dice and Shavit[1] equipped with a global *version-clock*. The algorithm uses *commit-time locking* and a *version-clock* per application, implemented as a counter which is atomically incremented by every store transaction using *Compare-and-Swap*, and is read by every load transaction[2]. Much like the *TL* algorithm, each transactional location is associated with a *versioned write-lock*, a spinlock that is acquired by a *CAS* operation and is released by a *store*. A single bit is reserved to indicate whether the lock is taken, whereas the rest is reserved for the objects *version*, which is advanced upon each successful write[2]. The algorithm considers two types of transactions: *write transactions* and *low-cost read transactions*[2].

The execution of *write transactions* can be summarised as follows. First, the global version-clock is saved in a thread-local variable, which will later be used to validate the version field of an accessed object. Then, the transaction is executed speculatively while maintaining local read and write-sets. A transactional load, much like the *TL* algorithm[1], first checks using a Bloom filter whether the accessed location appears in its write-set, and if so, it loads the last version of the object. A notable difference compared to the TL algorithm is the use of pre and post-validation of the accessed object's lock, which checks whether the versioned lock is free and has not changed. Moreover, the version of the object is compared to the sampled global version-clock. If it is found that the version of the objects is larger than the global version-clock, indicating that the object had been written by another transaction, the transaction is aborted[2]. Following, the locks for the addresses in the write-set is acquired. If this succeeds, the global version-clock is advanced using a *Compare-and-Swap* operation and the returned value is saved again in a thread-local variable, otherwise, the transaction is stopped[2]. Then, the read-set is revalidated using the same approach described above. This ensures that the locations had not been written by other transactions while the locks of the write-set are acquired and the global version-clock is advanced. Finally, the transaction *commits* and its held locks are released using a *store*[2].

Read-only transactions do not validate the read-set and therefore, can be executed efficiently[2]. Deciding whether a transaction is read-only can be done at compile-time, or can be checked by first running each transaction as read-only and when detecting a transactional write, unroll and retry as a writing transaction[2]. Read transactions' execution contains two steps. First, the global version-clock is sampled and saved in a thread-local variable. Then, the transaction is executed speculatively, while *post-validating* each load by checking whether the accessed version number is less or equal to the sampled clock. If not, the transaction aborts, else it commits[2].

4. Implementation

This section details the implementations proposed by the paper. This includes two Software Transactional Memory versions and transactional Red-Black Tree and Skiplist datastructures, which are used to benchmark and evaluate the performance of said STMs; moreover, they demonstrate the ease of designing transactional algorithms.

4.1 Software Transactional Memory

The lock-based Software Transactional Memory versions are implemented in C++ and they have the following interface which can be seen in Figure 4.1, that all implementations override. Upon creating a transaction `Tx`, `Tx.begin()` takes care of initialising its internal state. `Tx.commit()` attempts to commit the transaction and returns `true` if it succeeded and throws an `AbortException()` if it didn't. In the latter case, the transaction can be retried. `Tx.abort()` resets the transaction, clears its logs and, in case of an *encounter-order* transaction, rolls all writes back. `Tx.read(T *addr)` transactionally reads the value of the pointer `addr` and returns it, while `Tx.write(T *addr, T val)` transactionally writes in place of `addr` the value `val`.

```
template<class T>
struct Transaction {
    virtual void begin()          = 0;
    virtual void write(T *, T)    = 0;
    virtual T    read(T *)        = 0;
    virtual bool commit()         = 0;
    virtual void abort()          = 0;
};
```

Figure 4.1: Transactional interface

In pseudocode, statements that need to execute atomically are usually denoted with an `atomic` {...} block wrapping the instructions. Provided that compiler support exists for such a block, it could be parsed the following way: each *read* operation in the form `int a = *b;` is replaced by `int a = Tx.read(b);` and each write operation in the form `int b = 42;` is replaced by `Tx.write(&a, 42);`. Moreover, the atomic block needs to be able to repeat itself after seeing an inconsistent state and aborting. This is facilitated by a `try-catch` block nested inside a `while` loop. This code structure can be seen in Figure 4.2.

```
Transaction Tx;
bool done = false;
while (!done) {
    try {
        Tx.begin();

        /* atomic block */

        done = Tx.commit();
    }
    catch (AbortException&) {
        Tx.abort();
        done = false;
    }
}
```

Figure 4.2: Transactional code structure

The rest of this subsection details how transactions maintain consistent states through *ownership records*. Afterwards, the two proposed approaches to performing transactions is presented, which differ in terms of their lock-acquisition approach: one performs *encounter-order*, while the other performs *commit-time* locking. The implementations and their details are presented through how the algorithms override the `Transaction<T>` interface in Figure 4.1.

4.1.1 Ownership Records

When a transaction wishes to access a location for reading or writing, it first fetches the location's *ownership record (orec)* or *versioned write-lock*, as it is also described in [1, 2, 6]. Ownership records are a word in memory, which have two distinct states. When unlocked, the ownership record's lowermost bit is 0, and the rest of the bits represent the *version* of the object it references. Before writing an object, each transaction must acquire the record while ensuring that the object's version didn't change in the meantime. This can be achieved by an atomic *CAS*. When locked, the ownership record's lowermost bit is 1, while the rest of the bits represent the ID of the transaction. To unlock the record, its lowermost bit is cleared, while the rest of the bits store an increment of its previous version. Since every location in memory cannot be associated with a unique ownership record (since there would be too many), a many-to-one mapping is utilised to fetch the corresponding record. This is also called *lock-striping*[1, 2].

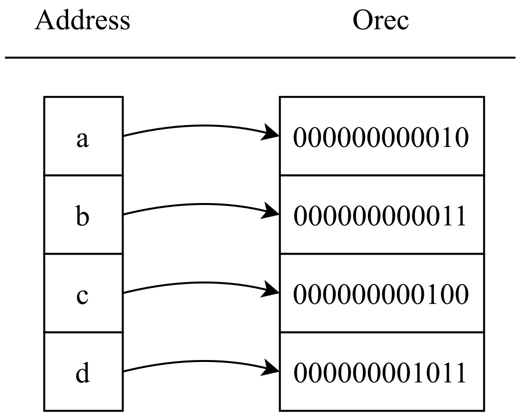

Figure 4.3: Addresses in memory with their associated orec. Address *a* is unlocked with version 1, Address *b* is locked by Transaction nr. 1, Address *c* is unlocked with version 2, and Address *d* is locked by Transaction nr. 5

4.1.2 Encounter-order Transactions

Encounter-order or *direct-update* transactions modify transactionally accessed memory directly. Between the transaction begins and tries to commit, all reads and writes are performed while keeping transaction-local *read*, and *undo* sets. The read set is used to validate during the execution of the transaction that all read-only locations' current version matches with the version recorded in the read set. The undo is used to roll back the writes upon aborting and restore the original values.

When encountering a transactional read, the location's *ownership record* is fetched. The transaction needs to first check whether it already holds the lock, and in that case, the location's value can be atomically loaded, while the current version of the object is stored in the transaction's read set. In case the transaction does not hold the associated lock, it checks whether some other transaction does. If it finds the location to be locked, it can either spin or abort and retry. If the location is not locked, the value can be atomically loaded, while the current version of the object is stored in the transaction's read set.

In the case of a transactional write, firstly, the ownership record associated with the address is fetched. The transaction then must acquire the record in order to ensure that only a single transaction can write to the same location. If the lock could not be acquired (because another transaction successfully acquired it in the meanwhile), the transaction aborts. If the transaction acquired the lock, or it already holds the record, the address' current value is saved in the transaction's *undo* set, and the new value is atomically stored at the address.

During the execution of the transaction, the read set must be validated to ensure that the recorded versions match the objects' current version. If a mismatch is found, there is a *read-write* conflict between two transactions. In that case, another transaction modified the object that the transaction previously read; therefore, the transaction with the invalid read-set must abort.

When committing, the transaction revalidates its read-set, and if it is found to be valid, releases all locks and returns. The transaction is now considered to be committed.

Upon aborting, the transaction uses its undo-set to roll back all writes and restore the previous state as if nothing had happened. Then, it can choose to retry immediately or utilise some form of back-off, which might be useful in case of high contention.

The encounter-order approach, similar to the approaches of [1, 11], makes committing much simpler. However, encounter-order locking has the downside of holding onto locks for a much longer time, potentially hindering performance under high contention.

4.1.3 Commit-time Transactions

Commit-time or *deferred-update* transactions utilise a higher level of *speculative execution* by deferring writes to commit time. During the execution, a *write-set* is built, which stores the to be written values. Naturally, a *read-set*, consisting of read locations and their current value, is also being book-kept, which ensures that the transaction sees valid states only.

A transactional read of an address first needs to check whether the address appears in its write-set as well, and in that case, return the latest new value in the write set. If the write set does not contain the address, the associated ownership record is fetched. If the location is found to be locked, the transaction can spin or abort. Else, the current version of the object is stored in the transaction's read set, and the location is atomically loaded.

All transactional writes are deferred to commit time; therefore, performing writes amounts to recording the address-value pair in the transaction's write-set.

During the execution of the transaction, the read-set is validated periodically by checking that the recorded version of objects matches the current version. If the read-set is found to be invalid, the transaction aborts.

When committing, all locks associated with entries in the write set must be acquired. This is done as follows: if a location in the write set appears in the read set as well (i.e. the transactional write depended on the read or vice-versa), the operation must atomically acquire the lock and validate that the version of the object at hand did not change during the execution of the transaction. If the location does not appear in the read set as well, the transaction simply acquires the associated locks. In case a lock cannot be acquired, the transaction aborts. Once all locks are acquired, the read-set is revalidated to ensure consistency, and finally, all updates are made visible using atomic stores. Afterwards, all locks are released, and the transaction is considered to be committed.

When aborting, the transaction must release all previously held locks and can be retried or can back-off based on contention.

The commit-time approach, similar to the approach of [1] and as also discussed in [1, 2, 6], presents a much easier write operation; however, due to its speculative nature, committing is much harder, and reading involves a look-aside into the write set. For the latter, fast approaches using *Bloom filters* exist, as discussed in [2].

Moreover, as will be presented in the Results section, commit-time transactions have an inherent limitation due to the way writes are handled. That is, only those writes will correctly be made visible, which do not depend on *previous* transactional writes. In the case of transactional red-black trees, inserting a node relies heavily on intermediate rotations and recolourings, which makes commit-time transactions unsuitable or the datastructure design unnatural. In the case of skiplists, the problem disappears since swinging pointers at a certain level does not depend on earlier computed results.

4.2 Transactional Red-Black Trees

One of the custom datastructures is a transactional Red-Black Tree implementation that is used to evaluate the performance of the proposed STM versions and demonstrate the ease of transactional datastructure design.

Binary trees are tree-graphs with a root in which each node has at most two children. A *binary search tree* (or BST for short) is a type of search datastructure built on top of binary trees. In a BST, each node is an object that contains `left-child`, `right-child` and `parent` pointers to other nodes, and a `key` attribute. Keys are stored with the condition that each node's key must be larger than any key in its left subtree and smaller than any key in its right sub-tree, or in other words, flattening the tree gives an increasing sequence of keys. In a binary search tree, the operations of search, insertion and deletion run in $\mathcal{O}(\log n)$ time in the average case; however, in the worst-case, these datastructures offer no greater performance than linked-lists with $\mathcal{O}(n)$ time[12]. The reason behind this is that BSTs do not *balance* themselves after the tree is modified; therefore, successive insertion of strictly smaller or larger elements will result in a linked-list-like structure starting at the root, which can be seen in Figure 4.4b.

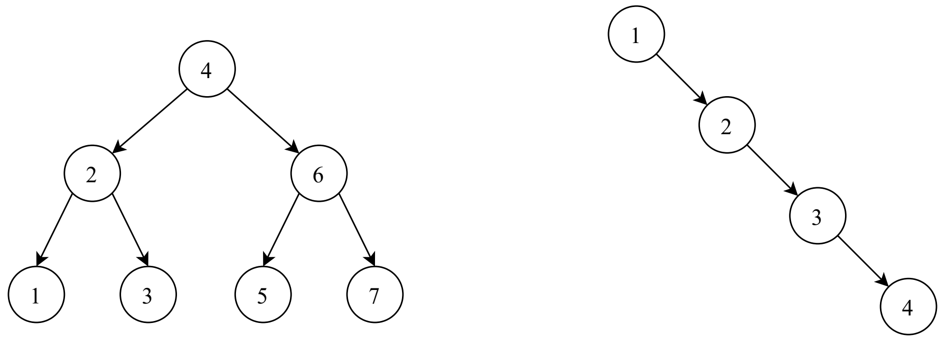

Figure 4.4: a) balanced BST b) unbalanced BST

Red-Black trees are a type of *self-balancing* binary search tree that aim to solve the previously mentioned performance issue, permitting $\mathcal{O}(\log n)$ worst-case search, insertion and deletion complexity[12]. A binary tree is considered to be balanced if the height of each nodes' left and right sub-tree differ by no more than one. The height of a binary tree is defined as the length of the longest possible path from the root to a leaf node (i.e. a node with no children).

In a red-black tree, compared to binary search trees, each node has an extra

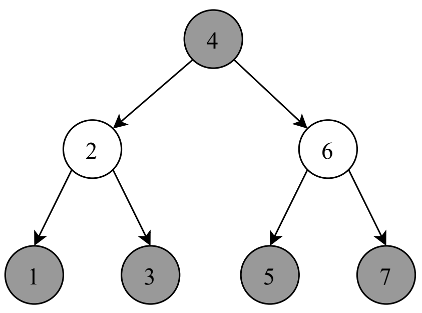

Figure 4.5: A red-black tree with shaded black and unshaded red nodes

attribute, its `colour`. Each node is coloured either red or black, which, along with some local rotation operations, ensure that the tree remains balanced after insertion and deletion. The tree is painted in such a way that the following properties hold[12]:

1. Every node is painted either red or black
2. The root and leaf nodes are black
3. Every red node's children must be black
4. From each node to its descendant leaves, all paths contain the same number of black nodes

In binary search trees, the root's parent and the leaf nodes' left and right child pointers point to `NIL` (i.e. to nothing). By convention, in a red-black tree T, a single sentinel node $T.nil$ is used to represent pointers to `NIL` which is always coloured black. This node has the same properties as any other node in the tree but with arbitrary key and left-right pointers. This implies that the leaf nodes of a red-black tree are always a single sentinel node; moreover, that the second part of property (2) is always ensured.

Inserting a key into a binary search tree can be summarised as follows. Start at the root of the tree and perform the following until the current node pointer is `NIL`: if the current node pointer is `NIL` insert the node at that position. If the current node is not null and the key we wish to insert is smaller than the current node's key, move the current node pointer to the left; else, move it to the right. This algorithm modifies the tree structure, which may or may not violate the properties of a red-black tree. These properties, which might be violated, are namely that the root needs to be black and that a red node cannot have red children. Therefore, when inserting keys into a red-black tree, the tree may need to be repainted, and the tree structure may need to be modified. Modifying the tree structure relies on local *rotations* which are illustrated in Figure 4.6.

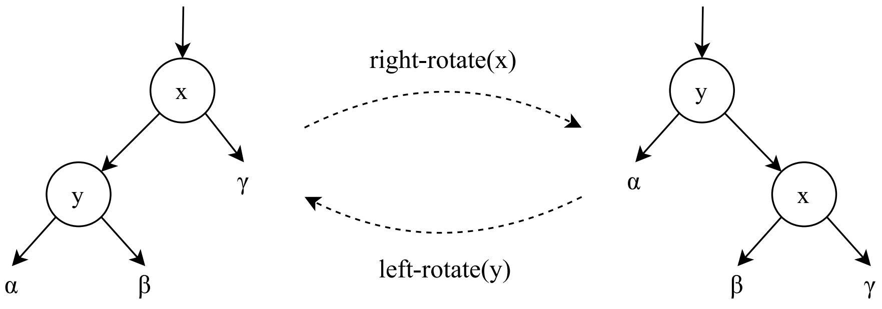

Figure 4.6: Rotation operations on binary search trees

One promising aspect of transactional datastructure design and one reason for choosing red-black trees is the simplicity involved in crafting them compared to the lock-based or lock-free approaches. Inserting and deleting keys in a red-black tree involves a *rebalancing* or *fixup* method compared to simple binary search trees, which makes the number of local rotations needed not known in advance. The lock-based approach would have difficulties creating a cycle-free acquisition order[6], and a lock-free approach[13, 14, 15] would involve modifying multiple locations at once. With transactions, the complexity of crafting such datastructure reduces to merely specifying which method needs to execute atomically, and the runtime environment will take care of the rest, which, arguably, also makes the process of crafting such datastructures more accessible and less prone to mistakes.

To demonstrate the simplicity regarding transactional datastructure design, consider as an example the insertion algorithm described in Algorithm 2, and its transactional translation in C++ in Figures 4.7. The algorithm, which inserts node z with key k into a red-black tree T, can be roughly summarised as follows, while, for the sake of simplicity, the description of the fixup method responsible for repainting and performing rotations on the tree is omitted. Firstly, pointers y and x are initialised to nil and the root of the tree in this order. While x does not point to nil, y is swung to x (this pointer keeps track of the parent of the new node z), and based on the new node z's key, x is swung either to its left or right child pointer. After the location of the new node is found, its parent pointer is assigned y. If the parent pointer is still nil, the tree was empty; therefore, the root is assigned the new node. If the tree was not empty, depending on the key of z, the parent node's left or right child is assigned z. Finally, the new node is initialised with nil left and right child pointers and the colour red.

In order to "transactionalise" this algorithm, one would need to specify which lines should execute atomically. In pseudocode, this is usually marked as an `atomic` {...} block wrapping the statements in the method. Provided that compiler support exists for such, the atomic block would be parsed with a transaction initialisation in the beginning and *read* statements in the form `int a = *b;` replaced with `int a = read(b);` and *write* statements in the form `A = b;` replaced with `write(&A, b);`. Here, the functions `T *read(T *)` and `void write(T *, T)` refer to the transactional interface described in Section 4.1. Such a translation of the sequential method can be seen in Figure 4.7.

The method `void tx_rb_insert(Node<T> *)` is a member of the class `TransactionalRBTree<T>` and operates on pointers to `Node<T>` objects. Notable differences to Algorithm 2 are line 2 which set up a new transaction as described in Section 4.1, line 6 which starts the transaction, and line 29 which commits it. The atomic block (i.e. statements that should execute atomically) is placed in a `try-catch` block which is retried until the transaction is able to commit. All reads are replaced by a method calls to `Tx.read()` and writes are replaced by calls to `Tx.write()`. In case the transaction fails, i.e. during the insertion procedure some inconsistencies are encountered, all methods throw an `AbortException` which is caught by the `catch` block. The transaction then aborts itself and is retried.

Algorithm 2 RB-Insert(T, z)

```
 1: y ← T.nil
 2: x ← T.root
 3: while x ≠ T.nil do
 4:     y ← x
 5:     if z.key < x.key then
 6:         x ← x.left
 7:     else
 8:         x ← x.right
 9:     end if
10: end while
11: z.p ← y
12: if y = T.nil then
13:     T.root ← z
14: else if z.key < y.key then
15:     y.left ← z
16: else
17:     y.right ← z
18: end if
19: z.left ← T.nil
20: z.right ← T.nil
21: z.color ← RED
22: RB-Insert-Fixup(T, z)
```

```
template<class T>
void tx_rb_insert(Node<T> *z) {
    Transaction Tx;
    bool done = false;
    while (!done) {
        try {
            Tx.begin();

            Node<T> *y = Tx.read(&nil), *n = y;
            Node<T> *x = Tx.read(&root);

            while (x != n) {
                y = x;
                if (z->key < Tx.read(&x->key))
                    x = Tx.read(&x->left);
                else
                    x = Tx.read(&x->right);
            }
            Tx.write(&z->parent, y);

            if (y == n)
                Tx.write(&root, z);
            else if (z->key < Tx.read(&y->key))
                Tx.write(&y->left, z);
            else Tx.write(&y->right, z);

            Tx.write(&z->left, nil);
            Tx.write(&z->right, nil);
            Tx.write(&z->colour, RED);

            tx_rb_insert_fixup(z);
            done = Tx.commit();
        }
        catch(AbortException&) {
            Tx.abort();
            done = false;
        }
    }
}
```

Figure 4.7: Transactional insertion into a Red-Black Tree

4.3 Transactional Skiplists

Skiplists are probabilistic datastructures similar in nature to balanced trees, proposed by William Pugh in 1990[16]. Skiplists can be thought of as linked-lists in which nodes are equipped with additional pointers to other nodes further into the list to allow for more efficient searching. They exhibit $\mathcal{O}$(log n) search, insert and delete average-case complexities and $\mathcal{O}$(n) worst-case complexities, similar to binary search trees.

Skiplists consist of linked nodes of a certain *height*, where each node with height h contains h forward pointers to other nodes. The height of a node is chosen with a certain probability p (usually chosen to be 1/2 or 1/4 etc.). In case when $p = 1/2$, 50% of nodes will have height 1, 25% of nodes will have height 2, etc. Skiplists consist of *levels* with level 0 being the uppermost one in Figure 4.8, and the last level being the bottom one, which essentially acts as a conventional linked-list. The maximum level is usually capped at $\log_{1/p}$ N, where N is the upper bound on the number of elements. This means that with $p = 1/2$, up to 2^{12} elements can be supported with the maximum level of 12[16]. Two dummy nodes, the *head* and *nil* represent the starting and ending points of all levels in a skiplist.

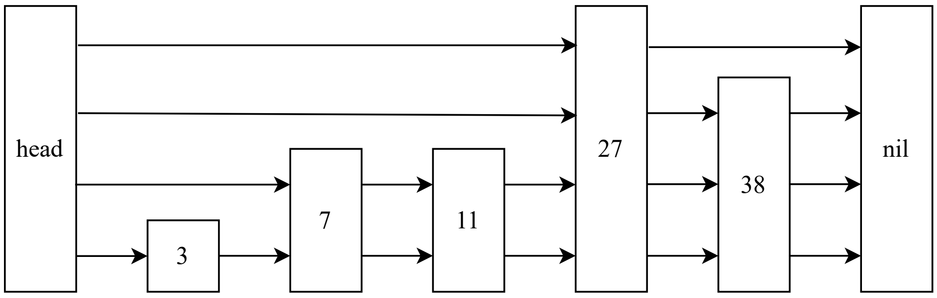

Figure 4.8: Skiplist with 5 elements

Searching in a skiplist can be summarised as follows. According to Pugh, starting at level $L(n) = \log_{1/p} n$ is most ideal, however, choosing level 0 adds only a small constant to the complexity[16]. Starting at some level n, the key that we are searching for is compared to the key of the first forward pointer of the current node. In case the keys match, the node is found. In case the key is smaller than the next node's, we move a level down; else, we move one to the right. Until the node is found, or the current node points to nil, this procedure repeats. This algorithm is highlighted using dotted arrows in Figure 4.9.

Inserting a key into a skiplist L, shown in Figure 4.9 and Algorithm 3, consists

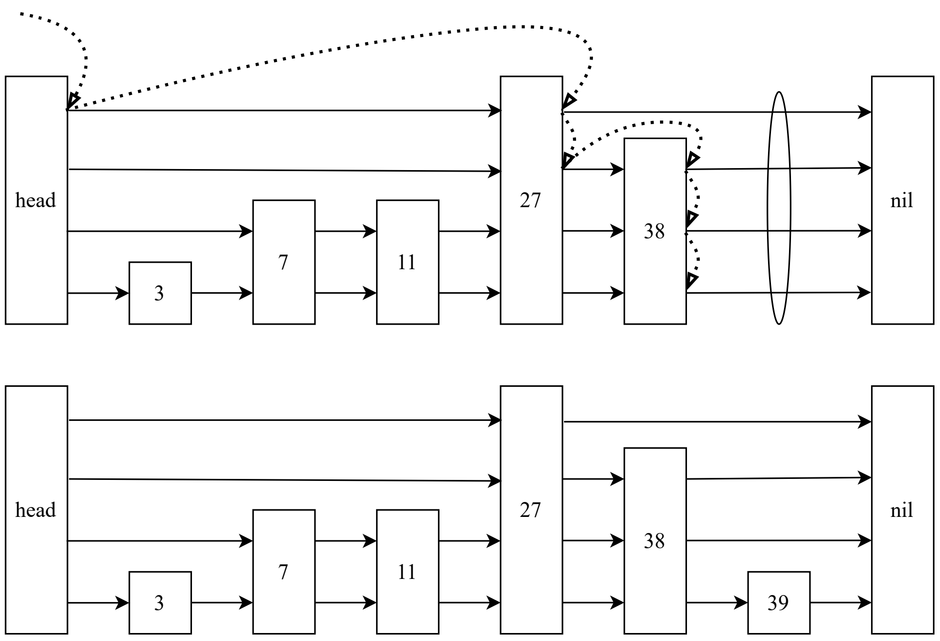

Figure 4.9: Inserting element 39 with height 1 into a skiplist. a) shows the search path to find the appropriate place on the last level. b) shows the final skiplist after insertion.

of two parts. Firstly, on lines 1-8, the appropriate place of the new node needs to be found while also building an update list of pointers, which will be used to *thread* the new node into the appropriate levels. Secondly, the new node needs to be inserted into the appropriate levels. If the node's random height is higher than the head's current one, the heads level is increased to match that of the new node. This is done on lines 11-17. Afterwards, a new node is allocated and threaded into the list. On line 20, the new nodes forward pointer at level i is assigned the ith element's forward pointer in the update array. Finally, on line 21, the update array's ith element's forward pointers are pointed to the new node.

Conceiving a lock-based or lock-free insertion algorithm into a skiplist would, much like with red-black trees, present a couple of difficulties that stem from the fact that the height of a node is not known in advance. A lock-based approach would need to figure out a cycle-free acquisition order of multiple node's forward pointers, and a lock-based approach would need to modify multiple locations at once when swinging the update array's forward pointers. This creates an unnatural and error-prone structure for the algorithms. Meanwhile, the transactional translation into C++, which shown in Figure 4.10, simplifies this into a naive sequential translation enclosed in a try-

Algorithm 3 Skiplist-Insert(L, key)

```
1: update ← [0..MaxLevel]
2: x ← L.head
3: for i ← L.level downto 0 do
4:     while x.forward[i] $\neq$ nil $\wedge$ x.forward[i].key $<$ key do
5:         x ← x.forward[i]
6:     end while
7:     update[i] ← x
8: end for
9: x ← x.forward[0]
10: if x = nil $\vee$ x.key $\neq$ key then
11:     h ← Get-Random-Height()
12:     if h $>$ L.level then
13:         for i ← L.level+1 to h do
14:             update[i] ← L.head
15:         end for
16:         L.level = h
17:     end if
18:     n ← Create-Node(h, key)
19:     for i ← 0 to level do
20:         n.forward[i] ← update[i].forward[i]
21:         update[i].forward[i] ← n
22:     end for
23: end if
```

catch block which is retried until the transaction can commit. All reads are replaced with calls to `Tx.read()` and all writes are replaced with calls to `Tx.write()`.

Presented here the function `void tx_skip_insert(Node<T> *n)` is a member of the class `TransactionalSkiplist<T>` and is responsible for inserting a Node of type `T`.

```
template<class T>
void tx_skip_insert(Node<T> *n) {
    Transaction<> Tx;
    bool done = false;
    while (!done) {
        try {
            Tx.begin();
            Node<T> *curr = Tx.read(&head);
            Node<T> *update[MAX_LEVEL + 1];
            for (int i = Tx.read(&level); i >= 0; i--) {
                Node<T> *curr = Tx.read(&curr->neighbours[i]);
                while (curr && Tx.read(&curr->value) < n->value)
                    curr = Tx.read(&curr->neighbours[i]);
                update[i] = curr;
            }
            curr = Tx.read(&curr->neighbours[0]);
            if (!curr || Tx.read(&curr->value) != n->value) {
                int h = n->height;
                if (h > Tx.read(&level)) {
                    for (int i = Tx.read(&level)+1; i<h+1; i++)
                        update[i] = Tx.read(&head);
                    Tx.write(&level, h);
                }
                for (int i = 0; i <= h; i++) {
                    Tx.write(&n->neighbours[i],
                                update[i]->neighbours[i]);
                    Tx.write(&update[i]->neighbours[i], n);
                }
            }
        }
        catch(AbortException&) {
            Tx.abort();
            done = false;
        }
    }
}
```

Figure 4.10: Transactional insertion into a Skiplist

5. Evaluation

5.1 Method

In order to benchmark and measure the performance of STM versions described in Section 4.1, certain iterations of tests will be performed on the datastructures detailed in Section 4.2 and 4.3.

The tests are run on the DAS5[17], a 6-cluster, 200-node, 16-core per node distributed system for scientific computing in the Netherlands, designed by the Advanced School of Computing and Imaging (ASCI). Of the whole network, a single node is utilised to run the test scripts. Therefore, the number of threads the tests are run on is bounded by 16, and the tests could be run on 1, 2, 4, 8 and 16 threads, respectively.

The first test, plotted in Figures 5.1, 5.2, involves 10k insertions into both the Red-Black Tree and Skiplist datastructures. In this test, only insertions are performed, as the procedure for deletion is much similar and involves about the same complexity and type of internal operations. The keys that each thread inserts are determined at runtime, where each thread inserts all integers in the interval $[i * n, (i + 1)n)$, where n is defined as the number of total insertions over the number of threads, and i is the id of the thread which ranges from 0 to 15 at the maximum. This results in storing 10k different values in the datastructure, with the actual insertion being high-contention. The same test was run using 100k and 1M insertions as well, and the results showed a very similar pattern to the 10k insertions test; therefore, it was decided to only include the latter.

In the case of the Red-Black Tree, only encounter-order transactions are utilised. Due to the nature and specific implementation details of the datastructure, the use of commit-time transactions proved unsuccessful. This is mainly due to how commit-time transactions perform their writes: namely that it records the *address* of the word in memory that it later wishes to change. Consider the scenario when the first node is inserted into the datastructure. The insertion algorithm described in Section 4.2 effectively performs two steps: it swings the root pointer to the new node and paints the root black. Let $\mathcal{A}$ be the address of the root, $\mathcal{B}$ be the address of the root's colour field, and finally, $\mathcal{Z}$ and $\mathcal{Z}'$ be the address of the to-be-inserted node and its colour. The write-list of the commit-time transaction upon committing this single insertion contains then the following 2-tuples: $\{\langle\mathcal{A}, \mathcal{Z}\rangle, \langle\mathcal{B}, \text{"}black\text{"}\rangle\}$. After a successful commit, however, one would find that the new root's colour is still red, and the last step, i.e. the recolouring of the root, affected the dummy node of the root and not the newly inserted node. When such dependencies of writes are included in some method, the proposed commit-time algorithm of Section 4.1.3 yields incorrect results. On careful inspection, one might notice that this is not a problem for the insertion algorithm of the Skiplist datastructure in Section 4.3, as inserting an element only swings pointers of nodes to the left of the newly inserted node, and those swings never depend on each other.

When performing the insertions, three measures are recorded: the mean wall-clock time of the total insertions with the standard deviation of the trial runs and the normalised speedup of each thread compared to the sequential run; finally, the mean abort rate, defined as the number of times a transaction needs to abort before successfully committing, is plotted.

In order to have a better understanding on the costs of the API operations of the transactional interface, the certain operations `begin()`, `read()`, `write()`, `commit()`, and `abort()`, are also separately timed and plotted in Figure 5.3. In this test, the mean wall-clock times of the operations involving 10k insertions into a Skiplist was measured and plotted in a scaled barplot.

5.2 Results

The 10k insertions into the Red-Black Tree datastructure gave interesting results. In the first column Figure 5.1, the mean wall-clock time of 10k insertions and its corresponding abort rates are plotted. As it can be seen, the execution time exhibits a logarithmic decrease until four threads, then a steady logarithmic increase as more than four threads are utilised. Similarly, for the abort rate, at $x = 4$, there is a clear inflexion point, where the slow increase in abort rate suddenly jumps, and at $x = 8$, it is about seven times as high. Due to the high abort rate and thereby slow execution time, the speedup compared to the sequential execution stops increasing at $x = 4$ and starts a drastic decrease as it can be seen on the first plot of Figure 5.2.

The maximum speedup that the encounter-order algorithm could achieve was about twice as fast as the sequential execution, which is similar to what the Skiplist insertion speedup achieved when the number of threads was 4.

In the case of the Skiplist, the results obtained are much more pleasing, as the second plot of the second column in Figure 5.1 exhibits a rather smooth logarithmic decrease and thereby optimal scaling properties for both algorithms. As the tests performed were high-contention, the commit-time algorithm performs about twice as slow compared to the encounter-order one. As can be seen, the abort rate of the commit-time algorithm is about five times as high as that of the encounter-order algorithm when 16 threads are utilised. The speedup, as it can be seen in the second plot of Figure 5.2, compared to the sequential execution of both algorithms are about the same, with execution on 16 threads achieving a speedup of about 5.8.

The relative execution times of the certain operations in the transactional API plotted in Figure 5.3, provides no surprises, as by analysing the complexities of the algorithms described in Sections 4.1.2 and 4.1.3, one could have already anticipated the results.

For the encounter-order algorithm, the `write()` operation clearly takes up more than half of all the combined execution times as it is responsible for read-set validation, lock acquisition and the atomic stores as well. Following it, the `commit()` method is the second most costly, responsible for the read-set validation and state reset. The `abort()` method takes up 7% of all execution times, which is responsible for atomically storing back the original values for the locations. The `begin()` and `read()` methods have negligible time complexities, comprised mostly of simple statements and an atomic load in case of the read method.

For the commit-time algorithm, as one could have expected, it is the `commit()` method that takes up most of the execution time. It is responsible for lock acquisition, which, due to the existence of the read-set, is in $\mathcal{O}(n^2)$; moreover, the read-set validation and atomic stores. Following are the `write()`, and `read()` methods, which both need to traverse the write-set to check whether the location is already contained and update the value or return the stored value, respectively. The methods for `begin()` and `abort()` have negligible impact on performance, as they consist mostly of simple statements of assignments.

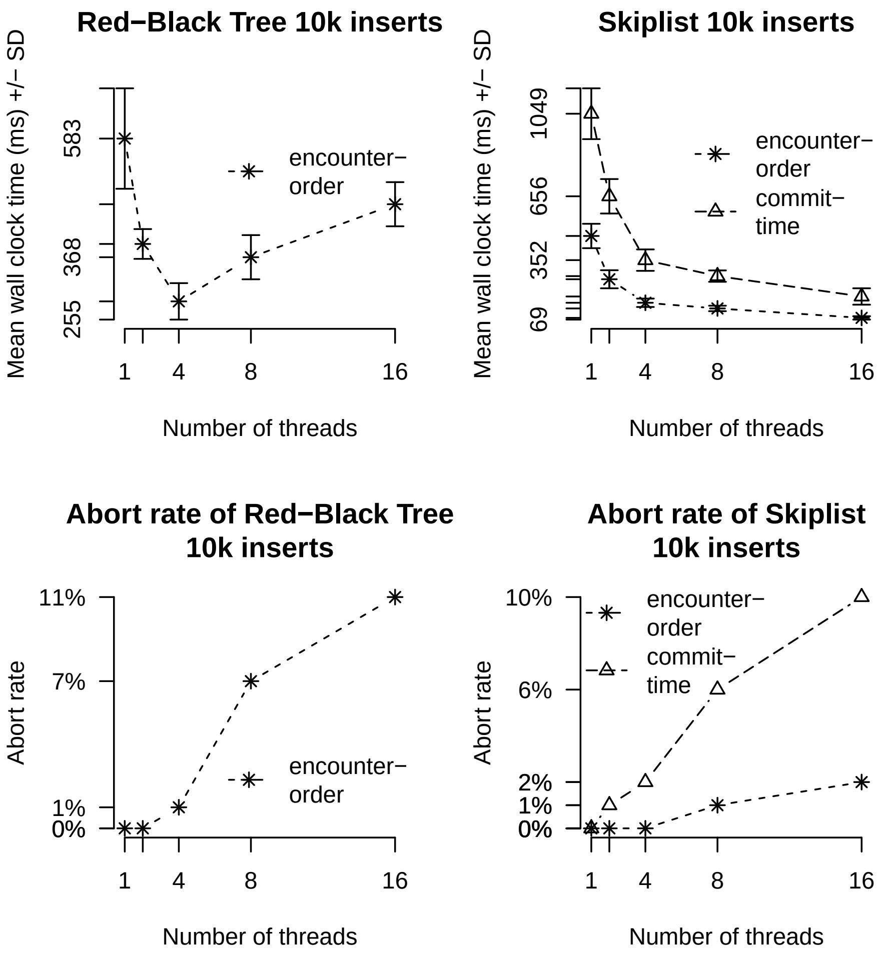

Figure 5.1: Red-Black Tree and Skiplist 10k insertion execution times using encounter-order and commit-time transactions and corresponding abort rates

5.3 Discussion

The performance of the Red-Black Tree is not ideal, as it does not scale well to highly concurrent applications. The suboptimal scaling properties can be largely attributed to the complexity of the Red-Black Tree insertion procedure, which potentially involves a large (and initially unknown) amount of rotations and recolouring, which increases to a great extent the probability of an invalid read-set and hence the rate of aborts. This can also be confirmed by looking at the second plot in the first column of Figure 5.1 where the abort

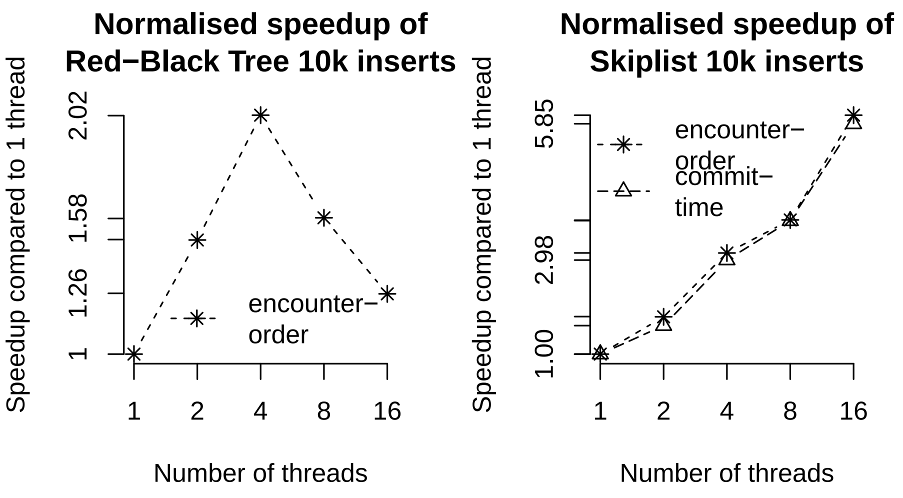

Figure 5.2: Red-Black Tree and Skiplist 10k insertions speedup compared to sequential execution time

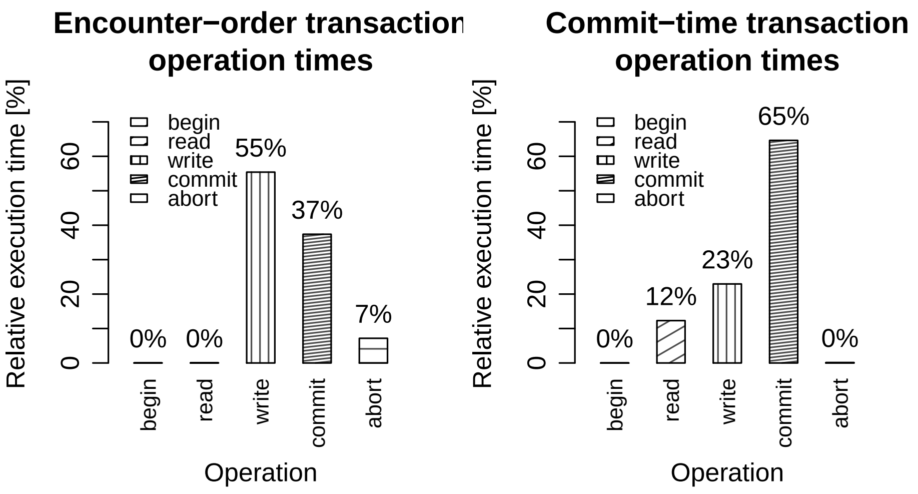

Figure 5.3: Relative operation times of encounter-order and commit-time transactions

rate of the encounter-order algorithm is plotted.

As the encounter-order algorithm described in Section 4.1.2 is similar in nature to the STM proposed by Ennals[11], these findings are in line with what is described by Dice and Shavit in their paper detailing the TL algorithm[1], where the performance of encounter-order algorithms are compared. In that paper, the Red-Black Tree test using Ennals' STM version exhibited very similar properties and patterns to what is described in the previous section, matching even the infliction point at $x = 4$.

In the case of the Skiplist, the encounter-order algorithm outperforms the commit-time with about a factor of two in mean wall-clock time and with a factor of 5 in abort rate, even though the achieved speedup was relatively similar for both. This is not in line with the findings of other papers like [1, 2], where it is concluded that commit-time locking should be utilised in case of high contention. However, it can easily be seen that in case of high contention, the abort rate of the commit-time algorithm would drastically increase since there is a very high chance that the read-set is invalidated. This can be further confirmed by looking at the last plot of the second column, where the abort rates are plotted. In the case of executing on 16 threads, there is a difference of a factor of 5 between the abort rate of the encounter order and commit-time locking mechanisms. Encounter-order locking is considered to be a disadvantage by [1, 2] as it holds onto locks for a much longer time. However, precisely because of this property, it has a much lower chance of aborting and could still achieve a better performance in the experiment. Moreover, the commit-time algorithm contains a lookaside into the write- and read-sets during committing, as explained in Section 4.1.3, which has $\mathcal{O}(n^2)$ complexity, and its read algorithm has linear time complexity. Compared to the encounter-order algorithm, which has $\mathcal{O}(1)$ read and linear time commit complexities, it is easy to see how it can overperform the commit-time version. We can conclude that the disadvantage of the encounter-order algorithm, namely that it holds onto locks for a much longer time, is still preferable to the complexities introduced in the commit-time algorithm in high-contention situations.

6. Conclusion

The paper set out to describe the intricacies of software transaction memory and its main approaches, namely the non-blocking and lock-based versions. In doing so, concurrent datastructure design emerged as a potential area where the transactional techniques could be applied successfully. By using the transactional approach to create concurrent datastructures, the process of crafting them moves away from the use of error-prone and difficult-to-reason-about lock-based and unnaturally structured lock-free algorithms to much more accessible and straightforward ones. Similarly to findings described in [18], this paper could also empirically conclude that apart from a few quirks in notation, writing concurrent transactional algorithms are not very different from their sequential counterpart while also being concurrent and correct. Results showed that the commit-time lock-based transactions could only be applied, without imposing an unnatural design on the datastructure itself, in cases where there are no direct dependencies between the writes. This is attributed to the fact that the commit-time transaction records its writes as a list of (address, value) 2-tuples that are oblivious to whether one write depends on the result of others. The encounter-order approach, however, can safely be applied in all situations; moreover, as opposed to findings of papers like [1, 2], under high-contention, it even outperforms its commit-time counterpart. This is due to the fact that under high contention, the commit-time transaction has a much higher abort rate as its read-set is invalidated more and more frequently, and its operations have worse time complexities, which hinders its performance. Further research could investigate how word-based commit-time transactions can overcome their limitations concerning the dependencies between their writes. Moreover, many popular datastructures whose internal complexities create rather complex lock-based or lock-free implementations could be explored and show that its transactional version is rather straightforward. Shifting to transactional design in place of the lock-based or lock-free style could be considered a viable alternative by creating an accessible abstraction to deal with concurrency.

Bibliography

[1] David Dice and Nir Shavit. "What Really Makes Transactions Faster?" In: *TRANSACT06 ACM Workshop* (Jan. 2006).

[2] Dave Dice, Ori Shalev, and Nir Shavit. "Transactional Locking II". In: DISC'06. Stockholm, Sweden: Springer-Verlag, 2006, pp. 194–208. ISBN: 3540446249. DOI: `10.1007/11864219_14`. URL: `https://doi.org/10.1007/11864219_14`.

[3] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.* AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: `10.1145/1465482.1465560`. URL: `https://doi.org/10.1145/1465482.1465560`.

[4] John L. Gustafson. "Reevaluating Amdahl's law". In: *Commun. ACM* (1988).

[5] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture.* ISCA '93. San Diego, California, USA: Association for Computing Machinery, 1993, pp. 289–300. ISBN: 0818638109. DOI: `10.1145/165123.165164`. URL: `https://doi-org.vu-nl.idm.oclc.org/10.1145/165123.165164`.

[6] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916.

[7] Nir Shavit and Dan Touitou. "Software Transactional Memory". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing.* PODC '95. Ottowa, Ontario, Canada: Association for Computing Machinery, 1995, pp. 204–213. ISBN: 0897917103. DOI: `10.1145/224964.224987`. URL: `https://doi.org/10.1145/224964.224987`.

[8] Maurice Herlihy et al. "Software Transactional Memory for Dynamic-Sized Data Structures". In: *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing.* PODC '03. Boston, Massachusetts: Association for Computing Machinery, 2003, pp. 92–101. ISBN: 1581137087. DOI: 10.1145/872035.872048. URL: https://doi.org/10.1145/872035.872048.

[9] K. Fraser. "Practical lock-freedom". In: PhD Thesis, Cambridge University Computer Laboratory, 2003.

[10] Keir Fraser and Tim Harris. "Concurrent Programming without Locks". In: *ACM Trans. Comput. Syst.* 25.2 (May 2007), 5–es. ISSN: 0734-2071. DOI: 10.1145/1233307.1233309. URL: https://doi.org/10.1145/1233307.1233309.

[11] Robert Ennals. "Software transactional memory should not be obstruction free". In: *In Intel Research Cambridge Tech Report.* 2006. DOI: 10.1.1.702.1468.

[12] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition.* 3rd. The MIT Press, 2009. ISBN: 0262033844.

[13] Jianwen Ma. "Lock-Free Insertions on Red-Black Trees". In: MSc thesis, University of Manitoba, Oct. 2003.

[14] Aravind Natarajan and Neeraj Mittal. "Fast Concurrent Lock-Free Binary Search Trees". In: *SIGPLAN Not.* 49.8 (Feb. 2014), pp. 317–328. ISSN: 0362-1340. DOI: 10.1145/2692916.2555256. URL: https://doi.org/10.1145/2692916.2555256.

[15] Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal. "Concurrent Wait-Free Red Black Trees". In: *Stabilization, Safety, and Security of Distributed Systems.* Ed. by Teruo Higashino et al. Springer International Publishing, 2013, pp. 45–60. ISBN: 978-3-319-03089-0.

[16] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: https://doi.org/10.1145/78973.78977.

[17] Henri Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (2016), pp. 54–63. DOI: 10.1109/MC.2016.127.

[18] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. "Is Transactional Programming Actually Easier?" In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP '10. Bangalore, India: Association for Computing Machinery, 2010, pp. 47–56. ISBN: 9781605588773. DOI: 10.1145/1693453.1693462. URL: https://doi.org/10.1145/1693453.1693462.