N^o d’ordre : ANNÉE 2012

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l’Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L’UNIVERSITÉ DE RENNES 1

Mention : Informátique

Ecole doctorale Matisse

présentée par

Tyler Crain

préparée à l’unité de recherche (n^o + nom abrégé)
(Nom développé de l’unité)
(Composante universitaire)

Intitulé de la thèse: Software Transactional Memory and Concurrent Data Structures: On the performance and usability of parallell programming abstractions

Thèse soutenue à l’INRIA – Rennes
le ?? Mars 2013

devant le jury composé de :

Software Transactional Memory and Concurrent Data Structures: On the performance and usability of parallel programming abstractions

Abstract

Writing parallel programs is well know difficult task and because of this there are some abstractions that can help make parallel programming easier. Two different approaches that try to solve this problem are transactional memory and concurrent data abstractions (such as sets or dictionaries). Transactional memory can be viewed as a methodology for parallel programming, allowing the programmer the ability to declare what sections of his code should be executed atomically, while concurrent data abstractions expose some specifically defined operations that can be safely executed concurrently to access and modify shared data. Importantly, even though these are separate solutions to the problem they are not mutually exclusive. They can be (and are) used together, as an example concurrent data abstractions are often used within transactions. Unfortunately they both are know to suffer from performance and (especially in the case of transactional memory) ease of use problems. This thesis examines and proposes some solutions trying to address these problems.

Transactional memory is designed to make parallel programming easier by allowing the programmer to define what blocks of code he wants to be executed atomically while leaving the difficult synchronization tasks for the underlying system. Even though the idea of a transaction is clear, how the programmer should interact with the abstraction is still not clearly defined and STM systems are known to suffer from performance problems. Three STM algorithms are presented to help study these problems.

1. One that satisfies two properties that can be considered good for efficiency.
2. One that guarantees transactions commit exactly once and hides the concepts of aborts to the programmer.
3. One that allows safe concurrent access to the same memory both inside and outside of transactions.

Concurrent data structures representing abstractions such as a set or map/dictionary are often an important component of parallel programs as they allow the programmer to access/store/and modify data safely and concurrently. Due to this they can be heavily used and highly contended in concurrent workloads leading to poor performance in programs. Some proposed solutions to this suggest weakening the abstraction, but this can make using the data structures more complicated. In this thesis a methodology is proposed that improves performance without weakening the abstraction. This methodology can also be applied to data structures used within transactions.

Acknowledgments

Contents

I Software Transactional Memory Algorithms **11**

1 Introduction **13**

2 Read Invisibility, Virtual World Consistency and Permissiveness are Compatible **15**

2.1 Related Work and Definitions . . . 15
2.2 Incompatibility Proof with Opacity . . . 15
2.3 Compatibility Proof with Virtual World Consistency . . . 15
2.4 A Probabilistic Permissive Virtual World Consistent STM Protocol with Invisible Reads . . . 15
2.4.1 Proof of Correctness . . . 15
2.4.2 Protocol Optimizations . . . 15
2.5 Introduction . . . 15
2.5.1 Software transactional memory (STM) systems . . . 15
2.5.2 Two consistency conditions for STM systems . . . 16
2.5.3 Desirable properties for STM systems . . . 17
2.5.4 Content of the paper . . . 18
2.6 STM computation model and base definitions . . . 18
2.6.1 Processes and atomic shared objects . . . 18
2.6.2 Transactions and object operations . . . 19
2.7 Opacity and virtual world consistency . . . 20
2.7.1 Base definitions . . . 20
2.7.2 Opacity and virtual world consistency . . . 21
2.8 Invisible reads, opacity and permissiveness are incompatible . . . 22
2.9 Step 1: Ensuring virtual world consistency with read invisibility . . . 23
2.9.1 Base objects, STM interface, incremental reads and deferred updates . . 23
2.9.2 The underlying data structures . . . 24
2.9.3 The $\mathsf{read}_T()$ and $\mathsf{write}_T()$ operations . . . 25
2.9.4 The $\mathsf{try\_to\_commit}_T()$ operation . . . 26
2.10 Step 2: adding probabilistic permissiveness to the protocol . . . 27
2.10.1 The IR_VWC_P protocol . . . 27
2.11 Conclusion . . . 28
2.12 Proof of the algorithm for VWC and read invisibility . . . 30
2.12.1 Proof that $\mathscr{C}$ is serializable . . . 30
2.12.2 Proof that the causal past of each aborted transaction is serializable . . 33
2.12.3 VWC and read invisibility . . . 34
2.13 Improving the base protocol described in Figure 2.4 . . . 35

2.13.1 Garbage collecting useless cells 35
2.13.2 Expediting read operations 36
2.13.3 From serializability to linearizability 36
2.14 Proof of the probabilistic permissiveness property 37
2.15 More on fast read operations 38
2.16 Making read operations invisible at commit time 40
2.17 Substraction on sets of intervals (line 20.H of Figure 2.6) 42
2.18 About the predicate of line 24.A of Figure 2.6 42

3 Universal Constructions and Transactional Memory 45
3.1 Universal Constructions 45
3.2 Aborts, Commits, and Liveness in Transactional Memory 45
3.3 A Universal Construction for Transaction Based Multi-Process Programs . . . 45
3.3.1 Proof of Linearizability 45
3.3.2 Proof of Liveness 45
3.3.3 Computation Complexity 45
3.4 ***An Implementation*** 45
3.5 Introduction 45
3.6 Related work 49
3.7 Computation models 51
3.7.1 The user programming model 51
3.7.2 The underlying system model 52
3.8 A universal construction for STM systems 53
3.8.1 Control variables shared by the processors 53
3.8.2 How the *t*-objects and *nt*-objects are represented 54
3.8.3 Behavior of a processor: initialization 55
3.8.4 Behavior of a processor: main body 55
3.8.5 Behavior of a processor: starvation prevention 58
3.9 Proof of the STM construction 59
3.10 The number of tries is bounded 65
3.11 A short discussion to conclude 66

4 Ensuring Strong Isolation in STM Systems 69
4.1 Isolation in STM Systems 69
4.2 A STM Protocol with Non-Blocking Non-Transactional Operations 69
4.2.1 Overview of TL2 69
4.2.2 The Protocol 69
4.2.3 ***Proof of Linearizability*** 69
4.3 ***An Implementation*** 69
4.4 Introduction 69
4.5 Correctness and Strong Isolation 70
4.6 A Brief Presentation of TL2 72
4.7 Implementing Terminating Strong Isolation 73
4.7.1 Memory Set-up and Data Structures. 74
4.7.2 Description of the Algorithm. 76
4.7.3 Non-transactional Operations. 76
4.7.4 Transactional Read and Write Operations. 77

4.8 Conclusion . . . 81
4.9 Version of algorithm that does not use NT-records . . . 81
4.10 Version of algorithm with non-blocking NT-reads and blocking NT-writes . . . 81

II Concurrent Data Structures 85

5 Introduction 87

6 Contention Friendly Data Structures 89
6.1 Non-Contention Friendly Data Structures . . . 89
6.2 The Contention Friendly Methodology . . . 89
6.2.1 In Transactional Memory . . . 89
6.3 Introduction . . . 89
6.4 Related Work . . . 91
6.5 The CF Methodology at a Glance . . . 92
6.5.1 Eager abstract modification . . . 93
6.5.2 Lazy structural adaptation . . . 93
6.5.3 Selective removal . . . 94
6.5.4 Avoiding contention during traversal . . . 95
6.6 Putting the CF Methodology to Work . . . 95
6.6.1 CF Skip list . . . 95
6.6.2 CF Tree . . . 96
6.6.3 CF Hash table . . . 97
6.7 Evaluation . . . 98
6.8 Conclusion . . . 99
6.9 Transactional Contention-Friendly Algorithms . . . 102
6.10 Pseudo-code and Description . . . 104
6.10.1 Tree and Skip List . . . 104
6.10.2 Abstract Operations . . . 107
6.10.3 Hash Table . . . 109
6.11 Correctness . . . 111
6.11.1 Skip List . . . 111
6.11.2 Tree . . . 112
6.11.3 HashTable . . . 113
6.12 Garbage Collection . . . 113
6.13 Future Work . . . 114
6.13.1 Lock-freedom . . . 114
6.13.2 Structural Adaptation Throttling . . . 115
6.13.3 Distributed Structural Adaptation . . . 115

7 A Contention Friendly Binary Search Tree 117
7.1 Lock Based Algorithm . . . 117
7.2 ***Lock Free Algorithm*** . . . 117
7.3 Transactional Memory Based Algorithm . . . 117
7.4 Introduction . . . 117
7.5 The Problem with Balanced Trees . . . 119

7.6 The Speculation-Friendly Binary Search Tree 122
7.6.1 Decoupling the tree rotation 122
7.6.2 Decoupling the node deletion 125
7.6.3 Optional improvements 126
7.6.4 Garbage collection 129
7.7 Correctness Proof 130
7.8 Experimental Evaluation 137
7.8.1 Testbed choices 137
7.8.2 Biased workloads and the effect of restructuring 138
7.8.3 Portability to other TM algorithms 138
7.8.4 Reusability for specific application needs 139
7.8.5 The vacation travel reservation application 139
7.9 Related Work 140
7.10 Conclusion 141

8 A Contention Friendly Skip-List 145
8.1 Lock Based Algorithm 145
8.2 Lock Free Algorithm 145
8.3 Transactional Memory Based Algorithm 145
8.4 Introduction 145
8.5 Related Work 146
8.6 Towards Contention-Friendliness 147
8.6.1 Eager abstract modification 148
8.6.2 Lazy selective structural adaptation 148
8.7 The Non-Blocking Skip List 149
8.7.1 Abstract operations 150
8.7.2 Structural adaptation 151
8.7.3 Non-Blocking 152
8.7.4 Garbage collection 153
8.7.5 Distributing the structural adaptation 153
8.8 Evaluation 154
8.9 Conclusion 155
8.10 Additional Evaluation 155
8.11 Correctness 160

9 A Contention Friendly Hash-Table 167
9.1 Lock Free Algorithm 167
9.2 Transactional Memory Based Algorithm 167

Conclusion 169

Bibliography 171

List of Publications 181

List of Figures

2.1 Examples of causal pasts 29
2.2 Invisible reads, opacity and permissiveness are incompatible 29
2.3 List implementing a transaction-level shared object X 30
2.4 Algorithm for the operations of the protocol 31
2.5 Commit intervals 32
2.6 Algorithm for the try_to_commit() operation of the permissive protocol 32
2.7 The cleaning background task BT 36
2.8 Fast read: Code to replace line 20 of Figure 2.4 39
2.9 Example of fast reads that would violate opacity 40
2.10 Subtraction on sets of intervals 42
2.11 Predicate of line 24.A of Figure 2.6 42

3.1 Initialization for processor P_x $(1 \leq x \leq m)$ 55
3.2 Algorithm for processor P_x $(1 \leq x \leq m)$ 57
3.3 The procedure select_next_process() 58
3.4 Procedure prevent_endless_looping() 59

4.1 Left: *Containment* (operation R_x should not return the value written to x inside the transaction). Right: *Non-Interference* (wile it is still executing, transaction T_1 should not have access to the values that were written to x and y by process p_2). 72
4.2 The memory set-up and the data structures that are used by the algorithm. . . . 74
4.3 Non-transactional operations for reading and writing a variable. 77
4.4 Transactional operations for reading and writing a variable. 78
4.5 Transaction begin/commit. 79
4.6 Transactional helper operations. 80
4.7 Non-transactional operations for reading and writing a variable. 81
4.8 Transactional operations for reading and writing a variable. 82
4.9 Transaction commit. 83
4.10 Non-transactional operations for reading and writing a variable. 83

6.1 Impact of contention on the performance of two 512-sized data structures (with 48 cores running with an increasing update ratio) 90
6.2 Inserting horizontally in the skip list 93
6.3 Adapting vertically the skip list structure 94
6.4 Tolerance to contention of Contention-Friendly (CF) and non-CF data structures (performance slowdown with respect to 0% updates) 99
6.5 Performance of the Contention-Friendly (CF) and non-CF data structures . . . 101

6.6 STM Speedup when using CF data structures instead of their existing counterparts on the Niagara 2 machine . . . 103

7.1 A balanced search tree whose complexity, in terms of the amount of accessed elements, is **(left)** proportional to h in a pessimistic execution and **(right)** proportional to the number of restarts in an optimistic execution . . . 118
7.2 The classical rotation modifies node j in the tree and forces a concurrent traversal at this node to backtrack; the new rotation left j unmodified, adds j' and postpones the physical deletion of j . . . 120
7.3 Comparing the AVL tree (AVLtree), the red-black tree (RBtree), the no-restructuring tree (NRtree) against the speculation-friendly tree (SFtree) on an integer set micro-benchmark with from 5% **(top)** to 20% updates **(bottom)** under normal **(left)** and biased **(right)** workloads . . . 142
7.4 The speculation-friendly library running with **(left)** another TM library ($\mathscr{E}$-STM) and with **(right)** the previous TM library in a different configuration (TinySTM-ETL, i.e., with eager acquirement) . . . 143
7.5 Elastic transaction comparison and reusability . . . 143
7.6 The speedup (over single-threaded sequential) and the corresponding duration of the vacation application built upon the red-black tree (RBtree), the optimized speculation-friendly tree (Opt SFtree) and the no-restructuring tree (NRtree) on **(left)** high contention and **(right)** low contention workloads, and with **(top)** the default number of transaction, **(middle)** $8\times$ more transactions and **(bottom)** $16\times$ more transactions . . . 144

8.1 Decoupling the eager abstraction insertion from the lazy selective adaptation . . 148
8.2 Skip list structure . . . 150
8.3 Comparison of our contention-friendly non-blocking skip list against the JDK concurrent skip list (ConcurrentSkipListMap) . . . 153
8.4 Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements . 155
8.5 Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements . 156
8.6 Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements 157
8.7 Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements 158
8.8 Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using the grow and shrink benchmark . . . 158

Part I

Software Transactional Memory Algorithms

Chapter 1

Introduction

Chapter 2

Read Invisibility, Virtual World Consistency and Permissiveness are Compatible

2.1 Related Work and Definitions

2.2 Incompatibility Proof with Opacity

2.3 Compatibility Proof with Virtual World Consistency

2.4 A Probabilistic Permissive Virtual World Consistent STM Protocol with Invisible Reads

2.4.1 Proof of Correctness

2.4.2 Protocol Optimizations

2.5 Introduction

2.5.1 Software transactional memory (STM) systems

The aim of an STM system is to simplify the design and the writing of concurrent programs by discharging the programmer from the explicit management of synchronization entailed by concurrent accesses to shared objects. This means that, when faced to synchronization, a programmer has to concentrate on where atomicity is required and not on the way it is realized.

More explicitly, an STM is a middleware approach that provides the programmers with the *transaction* concept [78, 57]. This concept is close but different from the notion of transactions encountered in databases [36, 10, 44]. A process is designed as (or decomposed into) a sequence of transactions, each transaction being a piece of code that, while accessing any number of shared objects, always appears as being executed atomically. The job of the programmer is only to define the units of computation that are the transactions. He does not have to worry about the fact that the objects can be concurrently accessed by transactions. Except when he defines the beginning and the end of a transaction, the programmer is not concerned by synchronization. It is the job of the STM system to ensure that transactions execute as if they were atomic.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is irrelevant from an efficiency point of view. So, a STM system has to do "its best" to execute and commit as many transactions per time unit as possible. Similarly to a scheduler, a STM system is an on-line algorithm that does not know the future. If the STM is not trivial (i.e., it allows several transactions that access the same objects in a conflicting manner to run concurrently), this intrinsic limitation can direct it to abort some transactions in order to ensure both transaction atomicity and object consistency. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered a new transaction). Abort is the price that has to be paid by transactional systems to cope with concurrency in absence of explicit synchronization mechanisms (such as locks or event queues).

2.5.2 Two consistency conditions for STM systems

The opacity consistency condition The classical consistency criterion for database transactions is serializability [157] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that commit. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting.

In contrast to database transactions that are usually produced by SQL queries, in a STM system the code encapsulated in a transaction is not restricted to particular patterns. Consequently a transaction always has to operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b-c$ is different from 0 in all consistent states (intuitively, a consistent state is a global state that, considering only the committed transactions, could have existed at some real time instant). If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b=c$ (and $b=c$ defines an inconsistent state). If this occurs, the transaction throws an exception that has to be handled by the process that invoked the corresponding transaction. Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops. Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction has to see always a consistent state of the data it accesses. The aborted transactions have to be harmless.

Informally suggested in [99], and formally introduced and investigated in [9], the *opacity* consistency condition requires that no transaction reads values from an inconsistent global state where, considering only the committed transactions, a *consistent global state* is defined as the state of the shared memory at some real time instant. Let us associate with each aborted transaction T its execution prefix (called *read prefix*) that contains all its read operations until T aborts (if the abort is entailed by a read, this read is not included in the prefix). An execution of a set of transactions satisfies the *opacity* condition if (i) all committed transactions plus each aborted transaction reduced to its read prefix appear as if they have been executed sequentially and (ii) this sequence respects the transaction real-time occurrence order.

Virtual world consistency This consistency condition, introduced in [152], is weaker than opacity while keeping its spirit. It states that (1) no transaction (committed or aborted) reads values from an inconsistent global state, (2) the consistent global states read by the committed transactions are mutually consistent (in the sense that they can be totally ordered) but (3) while the global state read by each aborted transaction is consistent from its individual point of view, the global states read by any two aborted transactions are not required to be mutually consistent. Said differently, virtual world consistency requires that (1) all the committed transactions be serializable [157] (so they all have the same "witness sequential execution") or linearizable [135] (if we want this witness execution to also respect real time) and (2) each aborted transaction (reduced to a read prefix as explained previously) reads values that are consistent with respect to its causal past only.

As two aborted transactions can have different causal pasts, each can read from a global state that is consistent from its causal past point of view, but these two global states may be mutually inconsistent as aborted transactions have not necessarily the same causal past (hence the name *virtual world* consistency). This consistency condition can benefit many STM applications as, from its local point of view, a transaction cannot differentiate it from opacity.

In addition to the fact that it can allow more transactions to commit than opacity, one of the main advantages of virtual world consistency lies in the fact that, as opacity, it prevents bad phenomena (as described previously) from occurring without requiring all the transactions (committed or aborted) to agree on the very same witness execution. Let us assume that each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.) when, executed alone, it reads values from a consistent global state. As, due to the virtual world consistency condition, no transaction (committed or aborted) reads from an inconsistent state, it cannot behave incorrectly despite concurrency, it can only be aborted. This is a first class requirement for transactional memories.

2.5.3 Desirable properties for STM systems

Invisible read operation A read operation issued by a transaction is *invisible* if it does not entail the modification of base shared objects used to implement the STM system [18]. This is a desirable property for both efficiency and privacy.

Base operations and underlying locks The use of expensive base synchronization operations such as
Compare&Swap() or the use of underlying locks to implement an STM system can make it inefficient and prevent its scalability. Hence, an STM systems should use synchronization operations sparingly (or even not at all) and the use of locks should be as restricted as possible.

Disjoint access parallelism Ideally, an STM system should allow transactions that are on distinct objects to execute without interference, i.e., without accessing the same base shared variables. This is important for efficiency and restricts the number or unnecessary aborts.

Permissiveness The notion of permissiveness has been introduced in [38] (in some sense, it is a very nice generalization of the notion of *obligation* property [15]). It is on transaction abort. Intuitively, an STM system is *permissive* "if it never aborts a transaction unless necessary for correctness" (otherwise it is *non-permissive*). More precisely, an STM system is permissive with respect to a consistency condition (e.g., opacity) if it accepts every history that satisfies the condition.

Some STM systems are randomized in the sense that the commit/abort point of a transaction depends on a random coin toss. Probabilistic permissiveness is suited to such systems. A randomized STM system is *probabilistically permissive* with respect to a consistency condition if every history that satisfies the condition is accepted with positive probability [38].

As indicated in [38], an STM system that checks at commit time that the values of the objects read by a transaction have not been modified (and aborts the transaction if true) cannot be permissive with respect to opacity.

2.5.4 Content of the paper

This paper is on permissive STM systems with invisible reads. It has several contributions.

- It first shows that an STM system that satisfies read invisibility and opacity cannot be permissive.
- The paper then presents an STM system (called IR_VWC_P) that satisfies read invisibility, virtual world consistency and permissiveness. The STM IR_VWC_P protocol presents additional noteworthy properties.
  - It uses only base read/write operations and locks, each associated with a shared object. Moreover, a lock is used at most once by a transaction at the end of a transaction (when it executes an operation called try_to_commit()).
  - it satisfies the disjoint access parallelism property.

The paper is made up of 8.9 sections. Section 2.6 presents the computation model. Section 2.7 presents the opacity and virtual world consistency conditions. Section 2.8 shows that opacity, read invisibility and permissiveness are incompatible. The IR_VWC_P protocol is then described incrementally. Section 2.9 presents first a base version that satisfies the VWC property and invisibility of read operations. Finally, Section 2.10 enriches it to obtain a permissive protocol. Section 8.9 concludes the paper.

2.6 STM computation model and base definitions

2.6.1 Processes and atomic shared objects

An application is made up of an arbitrary number of processes and m shared objects. The processes are denoted p_i, p_j, etc., while the objects are denoted $X, Y, \ldots$, where each id X is such that $X \in \{1, \cdots, m\}$. Each process consists of a sequence of transactions (that are not known in advance).

Each of the m shared objects is an atomic read/write object. This means that the read and write operations issued on such an object X appear as if they have been executed sequentially, and this "witness sequence" is legal (a read returns the value written by the closest write that precedes it in this sequence) and respects the real time occurrence order on the operations on X (if $op1(X)$ terminates before $op2(X)$ starts, $op1$ appears before $op2$ in the witness sequence associated with X).

2.6.2 Transactions and object operations

Transaction A transaction is a piece of code that is produced on-line by a sequential process (automaton), that is assumed to be executed atomically (commit) or not at all (abort). This means that (1) the transactions issued by a process are totally ordered, and (2) the designer of a transaction does not have to worry about the management of the base objects accessed by the transaction. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or write any shared object.

The set of the objects read by a transaction defines its *read set*. Similarly the set of objects it writes defines its *write set*. A transaction that does not write shared objects is a *read-only* transaction, otherwise it is an *update* transaction. A transaction that issues only write operations is a *write-only* transaction.

Transaction are assumed to b dynamically defined. The important point is here that the underlying STM system does not know in advance the transactions. It is an on-line system (as a scheduler).

Operations issued by a transaction We denote operations on shared objects in the following way. A read operation by transaction T on object X is denoted $X.\mathsf{read}_T()$. Such an operation returns either the value v read from X or the value *abort*. When a value v is returned, the notation $X.\mathsf{read}_T(v)$ is sometimes used. Similarly, a write operation by transaction T of value v into object X is denoted $X.\mathsf{write}_T(v)$ (when not relevant, v is omitted). Such an operation returns either the value *ok* or the value *abort*. The notations $\exists\, X.\mathsf{read}_T(v)$ and $\exists\, X.\mathsf{write}_T(v)$ are used as predicates to state whether a transaction T has issued a corresponding read or write operation.

If it has not been aborted during a read or write operation, a transaction T invokes the operation $\mathsf{try\_to\_commit}_T()$ when it terminates. That operation returns it *commit* or *abort*.

Incremental snapshot As in [3], we assume that the behavior of a transaction T can be decomposed in three sequential steps: it first reads data objects, then does local computations and finally writes new values in some objects, which means that a transaction can be seen as a software read_modify_write() operation that is dynamically defined by a process. (This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model.)

The read set is defined incrementally, which means that a transaction reads the objects of its read set asynchronously one after the other (between two consecutive reads, the transaction can issue local computations that take arbitrary, but finite, durations). We say that the transaction T computes an *incremental snapshot*. This snapshot has to be *consistent* which means that there is a time frame in which these values have co-existed (as we will see later, different consistency conditions consider different time frame notions).

If it reads a new object whose current value makes inconsistent its incremental snapshot, the transaction is directed to abort. If the transaction is not aborted during its read phase, T issues local computations. Finally, if the transaction is an update transaction, and its write operations can be issued in such a way that the transaction appears as being executed atomically, the objects of its write set are updated and the transaction commits. Otherwise, it is aborted.

Read prefix of an aborted transaction A read prefix is associated with every transaction that aborts. This read prefix contains all its read operations if the transaction has not been aborted during its read phase. If it has been aborted during its read phase, its read prefix contains all read operations it has issued before the read that entailed the abort. Let us observe that the values obtained by the read operations of the read prefix of an aborted transaction are mutually consistent (they are from a consistent global state).

2.7 Opacity and virtual world consistency

This section defines formally opacity [9] and virtual world consistency [152]. First, we define some properties of STM executions. Then, based on these definitions, opacity and virtual world consistency are defined.

2.7.1 Base definitions

Preliminary remark Some of the notions that follow can be seen as read/write counterparts of notions encountered in message-passing systems (e.g., partial order and happened before relation [17], consistent cut, causal past and observation [2, 23]).

Strong **transaction history** The execution of a set of transactions is represented by a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$, called *transaction history*, that states a structural property of the execution of these transactions capturing the order of these transactions as issued by the processes and in agreement with the values they have read. More formally, we have:

- PO is the set of transactions including all committed transactions plus all aborted transactions (each reduced to its read prefix).
- $T1 \rightarrow_{PO} T2$ (we say "$T1$ precedes $T2$") if one of the following is satisfied:
  1. Strong process order. $T1$ and $T2$ have been issued by the same process, with $T1$ first.
  2. Read_from order. $\exists\ X.\mathsf{write}_{T1}(v)\ \wedge\ \exists\ X.\mathsf{read}_{T2}(v)$. This is denoted $T1 \xrightarrow{X}_{rf} T2$. (There is an object X whose value v written by $T1$ has been read by $T2$.)
  3. Transitivity. $\exists T : (T1 \rightarrow_{PO} T) \wedge\ (T \rightarrow_{PO} T2)$.

Weak **transaction history** The definition of a weak transaction history is the same as the one of a strong transaction history except for the "process order" relation that is weakened as follows:

- Weak process order. $T1$ and $T2$ have been issued by the same process with $T1$ first, and $T1$ is a committed transaction.

This defines a less constrained transaction history. In a weak transaction history, no transaction "causally depends" on an aborted transaction (it has no successor in the partial order).

Independent transactions and sequential execution Given a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ that models a transaction execution, two transactions $T1$ and $T2$ are *independent* (or concurrent) if neither is ordered before the other: $\neg(T1 \rightarrow_{PO} T2)\ \wedge\ \neg(T2 \rightarrow_{PO} T1)$. An execution such that $\rightarrow_{PO}$ is a total order, is a *sequential* execution.

Causal past of a transaction Given a partial order $\widehat{PO}$ defined on a set of transactions, the *causal past* of a transaction T, denoted $past(T)$, is the set including T and all the transactions T' such that $T' \rightarrow_{PO} T$.

Let us observe that, when $\widehat{PO}$ is a weak transaction history, an aborted transaction T is the only aborted transaction contained in its causal past $past(T)$. Differently, in a strong transaction history, an aborted transaction always causally precedes the next transaction issued by the same process. As we will see, this apparently small difference in the definition of strong and weak transaction partial orders has a strong influence on the properties of the corresponding STM systems.

Linear extension A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ is a topological sort of this partial order, i.e.,

- $S = PO$ (same elements),
- $\rightarrow_S$ is a total order, and
- $(T1 \rightarrow_{PO} T2) \Rightarrow (T1 \rightarrow_S T2)$ (we say "$\rightarrow_S$ respects $\rightarrow_{PO}$").

Legal transaction The notion of legality is crucial for defining a consistency condition. It expresses the fact that a transaction does not read an overwritten value. More formally, given a linear extension $\widehat{S}$, a transaction T is *legal* in $\widehat{S}$ if, for each $X.\mathsf{read}_T(v)$ operation, there is a committed transaction T' such that:

- $T' \rightarrow_S T$ and $\exists\, X.\mathsf{write}_{T'}(v)$, and
- $\nexists\, T''$ such that $T' \rightarrow_S T'' \rightarrow_S T$ and $\exists\, X.\mathsf{write}_{T''}()$.

If all transactions are legal, the linear extension $\widehat{S}$ is legal.

In the following, a legal linear extension of a partial order, that models an execution of a set of transactions, is sometimes called a *sequential witness* (or witness) of that execution.

Real time order Let $\rightarrow_{RT}$ be the *real time* relation defined as follows: $T1 \rightarrow_{RT} T2$ if $T1$ has terminated before $T2$ starts. This relation (defined either on the whole set of transactions, or only on the committed transactions) is a partial order. In the particular case where it is a total order, we say that we have a real time-complying sequential execution.

A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ is real time-compliant if $\forall T, T' \in S$: $(T \rightarrow_{RT} T') \Rightarrow (T \rightarrow_S T')$.

2.7.2 Opacity and virtual world consistency

Both opacity and virtual world consistency ensures that no transaction reads from an inconsistent global state. If each transaction taken alone is correct, this prevents bad phenomena such as the ones described in the Introduction (e.g., entering an infinite loop). Their main difference lies in the fact that opacity considers strong transaction histories while virtual world consistency considers weak transaction histories.

Définition 2.1 *A strong transaction history satisfies the opacity consistency condition if it has a real time-compliant legal linear extension.*

Examples of protocols implementing the opacity property, each with different additional features, can be found in [99, 14, 152, 21].

Définition 2.2 *A weak transaction history satisfies the virtual world consistency condition if (a) all its committed transactions have a legal linear extension and (b) the causal past of each aborted transaction has a legal linear extension.*

A protocol implementing virtual world consistency can be found in [152] where it is also shown that any opaque history is virtual world consistent. In contrast, a virtual world consistent history is not necessarily opaque.

To give a better intuition of the virtual world consistency condition, let us consider the execution depicted on Figure 2.1. There are two processes: p_1 has sequentially issued T_1^1, T_1^2, T_1' and T_1^3, while p_2 has issued T_2^1, T_2^2, T_2' and T_2^3. The transactions associated with a black dot have committed, while the ones with a grey square have aborted. From a dependency point of view, each transaction issued by a process depends on its previous committed transactions and on committed transactions issued by the other process as defined by the read-from relation due to the accesses to the shared objects, (e.g., the label y on the dependency edge from T_2^1 to T_1' means that T_1' has read from y a value written by T_2^1). In contrast, since an aborted transaction does not write shared objects, there is no dependency edges originating from it. The causal past of the aborted transactions T_1' and T_2' are indicated on the figure (left of the corresponding dotted lines). The values read by T_1' (resp., T_2') are consistent with respect to its causal past dependencies.

2.8 Invisible reads, opacity and permissiveness are incompatible

Theorem 1 *Read invisibility, opacity and permissiveness (or probabilistic permissiveness) are incompatible.*

Proof Let us first consider permissiveness. The proof follows from a simple counter-example where three transactions T_1, T_2 and T_3 issue sequentially the following operations (depicted in Figure 2.2).

1. T_3 reads object X.
2. Then T_2 writes X and terminates. If the STM system is permissive it has to commit T_2. This is because if (a) the system would abort T_2 and (b) T_3 would be made up of only the read of X, aborting T_2 would make the system non-permissive. Let us notice that, at the time at which T_2 has to be committed or aborted, the future behavior of T_3 is not known and T_1 does not yet exist.
3. Then T_1 reads X and Y. Let us observe that the STM system has not to abort T_1. This is because when T_1 reads X there is no conflict with another transaction, and similarly when T_1 reads Y.
4. Finally, T_3 writes Y and terminates. let us observe that T_3 must commit in a permissive system where read operations (issued by other processes) are invisible. This is because, due to read invisibility, T_3 does not know that T_1 has previously issued a read of Y. Moreover, T_1 has not yet terminated and terminates much later than T_3. Hence, whatever the commit/abort fate of T_1, due to read invisibility, no information on the fact that T_1 has accessed Y has been passed from T_1 to T_3: when the fate of T_3 has to be decided, T_3 is not aware of the existence of T_1.

The strong transaction history $\widehat{PO} = (\{T_1, T_2, T_3\}, \rightarrow_{PO})$ associated with the previous execution is such that:

- $T_3 \rightarrow_{PO} T_2$ (follows from the fact that T_2 overwrites the value of X read by T_3).

- $T_2 \rightarrow_{PO} T_1$ (follows from the fact that T_1 reads the value of X written by T_2). Let us observe that this is independent from the fact that T_1 will be later aborted or committed. (If T_1 is aborted it is reduced to its read prefix "X.read(); Y.read()" that obtained values from a consistent global state.)
- Due to the sequential accesses on Y that is read by T_1 and then written by T_3, we have $T_1 \rightarrow_{PO} T_3$.

It follows from the previous item that $T_1 \rightarrow_{PO} T_1$. A contradiction from which we conclude that there is no protocol with invisible read operations that both is permissive and satisfies opacity.

Let us now consider probabilistic permissiveness. Actually, the same counter-example and the same reasoning as before applies. As none of T_2 and T_3 violates opacity, a probabilistic STM system that implements opacity with invisible read operations has a positive probability of committing both of them. As read operations are invisible, there is positive probability that both read operations on X and Y issued by T_1 be accepted by the STM system. It then follows that the strong transaction history $\widehat{PO} = (\{T_1, T_2, T_3\}, \rightarrow_{PO})$ associated with the execution in which T_2 and T_3 are committed while T_1 is aborted has a positive probability to be accepted. It is trivial to see that this execution is the same as in the non-probabilistic case for which it has been shown that this history is not opaque.

From this we have that read invisibility, permissiveness, and opacity are incompatible. $\Box_{Theorem\ 1}$

Remark 1 Let us observe that any opaque system with invisible reads would be required to abort T_3. When T_3 performs the try_to_commit() operation detecting that its read of X has been overwritten, it must abort (this is because T_3 has no way of knowing whether or not T_1's read exists at this point, so T_3 must abort in order to ensure safety). From this we have that read invisibility, permissiveness, and opacity are *incompatible* in the sense that any pair of properties can be satisfied only if the third is omitted.

Remark 2 The previous proof shows that opacity is a too strong consistency condition when one wants both read invisibility and permissiveness. Differently, when considering the previous execution, the virtual world consistency protocol IR_VWC_P presented in this paper will abort transaction T_1. It is easy to see that the corresponding weak transaction history is virtual world consistent: The read prefix "X.read$_{T_1}$(); Y.read$_{T_1}$()" of the aborted transaction T_1 can be ordered after T_2 (and T_3 does not appear in its causal past).

2.9 Step 1: Ensuring virtual world consistency with read invisibility

As announced in the Introduction, the protocol IR_VWC_P is built is two steps. This section presents the first step, namely, a protocol that ensures virtual consistency with invisible read operations. The second step (Section 2.10) will enrich this base protocol to obtain probabilistic permissiveness.

2.9.1 Base objects, STM interface, incremental reads and deferred updates

The underlying system on top of which is built the STM system is made up of base shared read/write variables (also called registers) and locks. Some of the base variables are used to contain pointer values. As we will see, not all the base registers are required to be atomic. There is an exclusive lock per shared object.

The STM system provides the process that issues a transaction T with four operations. The operations $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$ have been already presented. The operation $\mathsf{begin}_T()$ is invoked by a transaction T when it starts. It initializes local control variables.

The proposed STM system is based on the incremental reads and deferred update strategy. Each transaction T uses a local working space. When T invokes $X.\mathsf{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later $X.\mathsf{read}_T()$ invocations (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory may have changed in between. As already explained, this is the *incremental snapshot* strategy.

When T invokes $X.\mathsf{write}_T(v)$, it writes v into its working space (and does not access the shared memory) and always returns *ok*. Finally, if T is not aborted while it is executing $\mathsf{try\_to\_commit}_T()$, it copies the values written (if any) from its local working space to the shared memory. (A similar deferred update model is used in some database transaction systems.)

2.9.2 The underlying data structures

Implementing a transaction-level shared object Each transaction-level shared object X is implemented by a list. Hence, at the implementation level, there is a shared array $PT[1..m]$ such that $PT[X]$ is a pointer to the list associated with X. This list is made up of cells. Let $CELL(X)$ be such a cell. It is made up of the following fields (see Figure 2.3).

- $CELL(X).value$ contains the value v written into X by some transaction T.
- $CELL(X).begin$ and $CELL(X).end$ are two dates (real numbers) such that the right-open time interval $[CELL(X).begin..CELL(X).end[$ defines the lifetime of the value kept in $CELL(X).value$. Operationally, $CELL(X).begin$ is the commit time of the transaction that wrote $CELL(X).value$ and $CELL(X).end$ is the date from which $CELL(X).value$ is no longer valid.
- $CELL(X).last\_read$ contains the commit date of the latest transaction that read object X and returned the value $v = CELL(X).value$.
- $CELL(X).next$ is a pointer that points to the cell containing the first value written into X after $v = CELL(X).value$. $CELL(X).prev$ is a pointer in the other direction.

  It is important to notice that none of these pointers are used in the protocol (Figure 2.4) that ensures virtual world consistency and read invisibility. $CELL(X).next$ is required only when one wants to recycle e inaccessible cells (see Section 2.13.1). Differently, $CELL(X).next$ will be used to obtain permissiveness (see Section 2.10).

No field of a cell is required to be an atomic read/write register of the underlying shared memory. Moreover, all fields (but $CELL(X).last\_read$) are simple write-once registers. Initially $PT[X]$ points to a list made up of a single cell containing the tuple $\langle v_{init}, 0, +\infty, 0, \perp, \perp\rangle$, where v_{init} is the initial value of X.

Locks A exclusive access lock is associated with each read/write shared object X. These locks are used only in the $\mathsf{try\_to\_commit}()$ operation, which means that neither $X.\mathsf{read}_T()$ nor $X.\mathsf{write}_T()$ is lock-based.

Variables local to each process Each process p_i manages a local variable denoted $last\_commit_i$ whose scope is the entire computation. This variable (initialized to 0) contains the commit date associated with the last transaction committed by p_i. Its aim is to ensure that the transactions committed by p_i are serialized according to their commit order.

In addition to $last\_commit_i$, a process p_i manages the following local variables whose scope is the duration of the transaction T currently executed by process p_i.

- $window\_bottom_T$ and $window\_top_T$ are two local variables that define the time interval during which transaction T could be committed. This interval is $]window\_bottom_T..window\_top_T[$ (which means that its bounds do not belong to the interval). It is initially equal to $]last\_commit_i..+\infty[$. Then, it can only shrink. If it becomes empty (i.e., $window\_bottom_T \geq window\_top_T$), transaction T has to be aborted.
- lrs_T (resp., lws_T) is the read (resp., write) set of transaction T. Incrementally updated, it contains the identities of the transaction-level shared objects X that T has read (resp., written) up to now.
- $lcell(X)$ is a local cell whose aim is to contain the values that have been read from the cell pointed to by $PT[X]$ or will be added to that list if X is written by T. In addition to the six previous fields, it contains an additional field denoted $lcell(X).origin$ whose meaning is as follows. If X is read by T, $lcell(X).origin$ contains the value of the pointer $PT[X]$ at the time X has been read. If X is only written by T, $lcell(X).origin$ is useless.

Notation for pointers $PT[X]$, $cell(X).next$ and $lcell(X).origin$ are pointer variables. The following pointer notations are used. Let PTR be a pointer variable. $PTR\downarrow$ denotes the variable pointed to by PTR. Let VAR be a non-pointer variable. $\uparrow VAR$ denotes a pointer to VAR. Hence, $PTR \equiv \uparrow(PTR\downarrow)$ and $VAR \equiv (\uparrow VAR)\downarrow)$.

2.9.3 The $\mathsf{read}_T()$ and $\mathsf{write}_T()$ operations

When a process p_i invokes a new transaction T, it first executes the operation $\mathsf{begin}_T()$ which initializes the appropriate local variables.

The $X.\mathsf{read}_T()$ operation The algorithm implementing $X.\mathsf{read}_T()$ is described in Figure 2.4. When p_i invokes this operation, it returns the value locally saved in $lcell(X).value$ if $lcell(X)$ exists (lines 02 and 13). If $lcell(X)$ has not yet been allocated, p_i does it (line 03) and updates its fields *value*, *begin* and *origin* with the corresponding values obtained from the shared memory (lines 04-07). Process p_i then updates $window\_bottom_T$ and $window\_top_T$. These updates are as follows.

- The algorithm defines the commit time of transaction T as a point of the time line such that T could have executed all its read and write operations instantaneously as that time. Hence, T cannot be committed before a committed transaction T' that wrote the value of a shared object X read by T. According to the algorithm implementing the $\mathsf{try\_to\_commit}_T()$ operation (see line 27), the commit point of such a transaction T' is the time value kept in $lcell(X).begin$. Hence, p_i updates $window\_bottom_T$ to $\max(window\_bottom_T, lcell(X).begin)$ (line 08). X is then added to $lrst_T$ (line 09).
- Then, p_i updates $window\_top_T$ (the top side of T's commit window, line 10). If there is a shared object Y already read by T (i.e., $Y \in lrs_T$) that has been written by some other

transaction T'' (where T'' is a transaction that wrote Y after T read Y), then $window\_top_T$ has to be set to $commit\_time_{T''}$ if $commit\_time_{T''} < window\_top_T$. According to the algorithm implementing the $\mathsf{try\_to\_commit}_T()$ operation, the commit point of such a transaction T'' is the date kept in $(lcell(Y).origin \downarrow).end$. Hence, for each $Y \in lrs_T$, p_i updates $window\_bottom_T$ to $\min\big(window\_top_T, (lcell(Y).origin \downarrow).end$ (line 10).

Then, if the window becomes empty, the $X.\mathsf{read}_T()$ operation entails the abort of transaction T (line 11). If T is not aborted, the value written by T' (that is kept in $lcell(X).value$) is returned (line 13).

The $X.\mathsf{write}_T(v)$ operation The algorithm implementing this operation is described at lines 14-17 of Figure 2.4. If there is no local cell associated with X, p_i allocates one (line 14) and adds X to lws_T (line 15). Then it locally writes v into $lcell(X).value$ (line 16) and return ok (line 17). Let us observe that no $X.\mathsf{write}_T()$ operation can entail the abort of a transaction.

2.9.4 The $\mathsf{try\_to\_commit}_T()$ operation

The algorithm implementing this operation is described in Figure 2.4 (lines 18-34). A process p_i that invokes $\mathsf{try\_to\_commit}_T()$ first locks all transaction-level shared objects X that have been accessed by transaction T (line 18). The locking of shared objects is done in a canonical order in order to prevent deadlocks.

Then, process p_i computes the values that define the last commit window of T (lines 19-20). The update of $window\_top_T$ is the same as described in the $\mathsf{read}_T()$ operation. The update of $window\_top_T$ is as follows. For each register Y that T is about to write in the shared memory (if T is not aborted before), p_i computes the date of the last read of Y, namely the date $(PT[Y] \downarrow).last\_read$. In order not to invalidate this read (whose issuing transaction has been committed), p_i updates $window\_bottom_T$ to $\max((PT[Y] \downarrow).last\_read, window\_bottom_T)$. If the commit window of T is empty, T is aborted (line 21). All locks are then released and all local cells are freed.

If T's commit window is not empty, it can be safely committed. To that end p_i defines T's commit time as a finite value randomly chosen in the current window $]window\_bottom_T..window\_top_T[$ (let us remind that the bounds are outside the window, line 22). This time function is such that no two processes obtain the same time value.

Then, before committing, p_i has to (a) apply the writes issued by T to the shared objects and (b) update the "last read" dates associated with the shared objects it has read.

a. First, for every shared object $X \in lws_T$, process p_i updates $(PT[X] \downarrow).overwrite$ with T's commit date (line 23). When all these updates have been done, for every shared object $X \in lws_T$, p_i allocates a new shared memory cell $CELL(X)$ and fills in the four fields of $CELL(X)$ (lines 25-28). Process p_i also has to update the pointer $PT[X]$ to its new value (namely $\uparrow CELL(X)$) (line 28).

b. For each register X that has been read by T, p_i updates the field $last\_read$ to the maximum of its previous value and $commit\_time_T$ (lines 30-32). (Actually, this base version of the protocol remains correct when $X \in lrs_T$ is replaced by $X \in (lrs_T \setminus lws_T)$. (As this improvement is no longer valid in the final version of the $\mathsf{try\_to\_commit}_T()$ algorithm described in Figure 2.6, we do not consider it in this base protocol.)

Finally, after these updates of the shared memory, p_i releases all its locks, frees the local cells it had previously allocated (line 33) and returns the value *commit* (line 34).

On the random selection of commit points It is important to notice that, choosing randomly commit points (line 22, Figure 2.4), there might be "best/worst" commit points for committed transactions, where "best point" means that it allows more concurrent conflicting transactions to commit. Random selection of a commit point can be seen as an inexpensive way to amortize the impact of "worst" commit points (inexpensive because it eliminates the extra overhead of computing which point is the best).

Proof of the algorithm for VWC and read invisibility Due to space limitations, the proof is not included here. It can be found in Section 2.12 of the Appendix and in [5].

Improving the base protocol described in Figure 2.4 The base protocol presented previously can be improved on the following three points: how the useless cells can be collected, how read operations can be made fast and how serializability can be replaced by linearizability. Due to space limitations, these improvements are not included here. They can be found in Section 2.13 of the Appendix and in [5].

2.10 Step 2: adding probabilistic permissiveness to the protocol

This section presents the final IR_VWC_P protocol that ensures virtual world consistency, read invisibility and probabilistic permissiveness. The first part describes the protocol while the second part proves its correctness.

2.10.1 The IR_VWC_P protocol

To obtain a protocol that additionally satisfies probabilistic permissiveness, only the operation $\mathsf{try\_to\_commit}_T()$ has to be modified. The algorithms implementing the operations $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T()$ are exactly the same as the ones described in Figure 2.4. The algorithm implementing the new version of the operation $\mathsf{try\_to\_commit}_T()$ is described in Figure 2.6. As we are about to see, it is not a new protocol but an appropriate enrichment of the previous $\mathsf{try\_to\_commit}_T()$ protocol.

A set of intervals for each transaction Let us consider the execution depicted in Figure 2.5 made up of three transactions: $T1$ that writes X, $T2$ that reads X and obtains the value written by $T1$, and $T3$ that writes X. When we consider the base protocol described in Figure 2.4, the commit window of $T3$ is $]commit\_time_{T2}..+\infty[$. As the aim is not to abort a transaction if it can be appropriately serialized, it is easy to see that associating this window to $T3$ is not the best choice that can be done. Actually $T3$ can be serialized at any point of the commit line as long as the read of X by $Y2$ remains valid. This means that the commit point of $T3$ can be any point in $]0..commit\_time_{T1}[\, \cup\,]commit\_time_{T2}..+\infty[$.

This simple example shows that, if one wants to ensure probabilistic permissiveness, the notion of continuous commit window of a transaction is a too restrictive notion. It has to be replaced by a set of time intervals in order valid commit times not to be a priori eliminated from random choices.

Additional local variables According to the previous discussion, two new variables are introduced at each process p_i. The set $commit\_set_T$ is used to contain the intervals in which T will be allowed to commit. To compute its final value, the set $forbid_T$ is used to store the windows in which T cannot be committed.

The enriched try_to_commit$_T()$ **operation** The new try_to_commit$_T()$ algorithm is described in Figure 2.6. In a very interesting way, this try_to_commit$_T()$ algorithm has the same structure as the one described in Figure 2.4. The lines with the same number are identical in both algorithms, while the number of the lines of Figure 2.4 that are modified are postfixed by a letter. The new/modified parts are the followings.

- Lines 20.A-20.I replace line 20 of Figure 2.4 that was computing the value of $window\_bottom_T$. These new lines compute instead the set of intervals that constitute $commit\_set_T$. To that end they suppress from the initial interval $]window\_bottom_T, window\_top_T[$, all the time intervals that would invalidate values read by committed transactions. This is done for each object $X \in lws_T$ (line 20.H; see Appendix 2.17 for an example). If $commit\_set_T$ is empty, the transaction T is aborted (line 21.A).
- The commit time of a transaction T is now selected from the intervals in $commit\_set_T$ (line 22.A).
- Line 23 of Figure 2.4 was assigning, for each $X \in lws_T$, its value to $(PT[X] \downarrow).end$, namely the value $commit\_time_T$. This is now done by the new lines 23.A-23.E. Starting from $PT[X]$, these statements use the pointer $prev$ to find the cell (let us denote it say CX) of the list implementing X whose field $CX.end$ has to be assigned the value $commit\_time_T$. Let us remember that $CX.end$ defines the end of the lifetime of the value kept in $CX.value$. This cell CX is the first cell (starting from $PT[X]$) such that $CX.begin < commit\_time_T$.
- Line 24 of Figure 2.4 assigned its new value to every object $X \in lws_T$. Now such an object X has to be assigned its new value only if $commit\_time_T > (PT[X] \downarrow).begin$. This is because when $commit\_time_T < (PT[X] \downarrow).begin$, the value v to be written is not the last one according to the serialization order. Let us remember that the serialization order, that is defined by commit times, is not required to be real time-compliant (which would be required if we wanted to have linearizability instead of serializability, see Section 2.13.3). An example is given in Appendix 2.18. Finally, the pointer $prev$ is appropriately updated (line 28.A). (Starting from $(PT[X] \downarrow).next$, these pointers allows for the traversal of the list implementing X.)

Proof of the probabilistic permissiveness property Due to space limitations, the proof is not included here. It can be found in Section 2.14 of the Appendix and in [5].

2.11 Conclusion

This paper has investigated the relation linking read invisibility, permissiveness and two consistency conditions, namely, opacity and virtual world consistency. It has shown that read invisibility, permissiveness and virtual world consistency are compatible. To that end an appropriate STM protocol has been designed and proved correct. Interestingly enough, this new STM protocol has additional noteworthy features: (a) it uses only base read/write operations and a lock per object that is used at commit time only and (b) satisfies the disjoint access parallelism property.

The proposed IR_VWC_P protocol uses multiple versions (kept in a list) of each shared object X. Multi-version systems for STM systems have been proposed several years ago [4] and have recently received a new interest, e.g., [1, 20]. In contrast to our work, none of these papers consider virtual wold consistency as consistency condition. Moreover, both papers consider a different notion of permissiveness called multi-version permissiveness that states that read-only transactions are never aborted and an update transaction can be aborted only when in conflict with other transactions writing the same objects. More specifically, paper [20] studies inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions. This paper presents also a protocol with visible read operations that recovers useless versions. Paper [1] shows that multi-version permissiveness can be obtained from single-version. The STM protocol it presents satisfies the disjoint access parallelism property, requires visible read operations and uses *k*-Compare&single-swap operations.

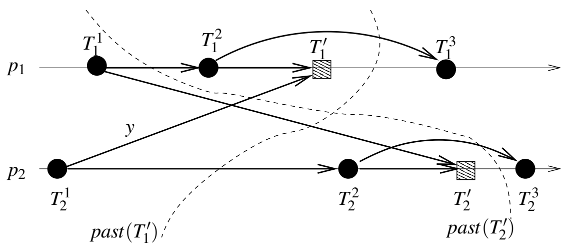


Figure 2.1: Examples of causal pasts

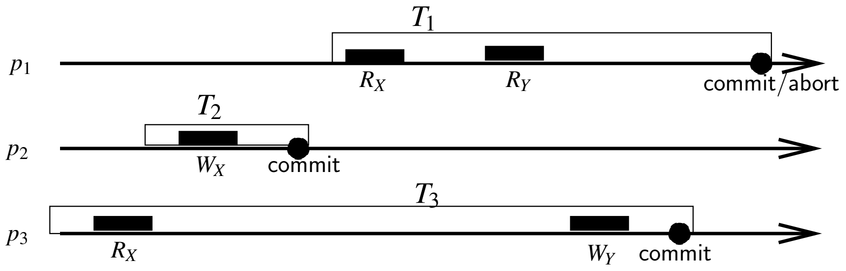


Figure 2.2: Invisible reads, opacity and permissiveness are incompatible

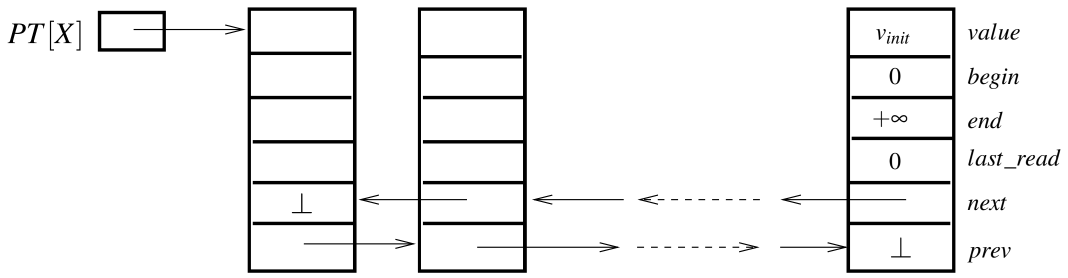


Figure 2.3: List implementing a transaction-level shared object X

2.12 Proof of the algorithm for VWC and read invisibility

Let $\mathscr{C}$ and $\mathscr{A}$ be the set of committed transactions and the set of aborted transactions, respectively. The proof consists of two parts. First, we prove that the set $\mathscr{C}$ is serializable. We then prove that that the causal past $past(T)$ of every transaction $T \in \mathscr{A}$ is serializable. In the following, in order to shorten the proofs, we abuse notations in the following way: we write "transaction T executes action A" instead of "the process that executes transaction T executes action A" and we use "$X.\mathsf{write}_T()$" as the predicate "T is a committed transaction and the operation $X.\mathsf{write}_T()$ belongs to the execution".

2.12.1 Proof that $\mathscr{C}$ is serializable

In order to show that $\mathscr{C}$ is serializable, we have to show that the partial order $\rightarrow_{PO}$ restricted to $\mathscr{C}$ accepts a legal linear extension. More precisely, we have to show that there exists an order $\rightarrow_S$ on the transactions of $\mathscr{C}$ such that the following properties hold:

1. $\rightarrow_S$ is a total order,
2. $\rightarrow_S$ respects the process order between transactions,
3. $\forall T1, T2 \in \mathscr{C} : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_S T2$ and,
4. $\forall T1, T2 \in \mathscr{C}, \forall X : (T1 \overset{X}{\rightarrow}_{rf} T2) \Rightarrow (\nexists T3 : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_S T3 \rightarrow_S T2)$.

In the following proof, $\rightarrow_S$ is defined according to the value of the *commit_time* variables of the committed transactions. If two transactions have the same *commit_time*, they are ordered according to the identities of the processes that issued them.

Lemma 1 *The order $\rightarrow_S$ is a total order.*

Proof The proof follows directly from the fact that $\rightarrow_S$ is defined as a total order on the commit times of the transactions of $\mathscr{C}$. $\Box_{Lemma\ 1}$

Lemma 2 *The total order $\rightarrow_S$ respects the process order between transactions.*

Proof Consider two committed transactions T and T' issued by the same process, T' being executed just after T. The variable $window\_bottom_{T'}$ of T' is initialized at $commit\_time_T$ and can only increase (during a read() operation at line 08, or during its try_to_commit() operation at line 20). Because $window\_bottom_{T'} > commit\_time_T$ (line 31), we have $T \rightarrow_S T'$. By transitivity, this holds for all the transactions issued by a process. $\Box_{Lemma\ 2}$

```
operation $\mathsf{begin}_T()$:
(01) $window\_bottom_T \leftarrow last\_commit_i$; $window\_top_T \leftarrow +\infty$; $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$.
==========================================================================
operation $X.\mathsf{read}_T()$:
(02) if ($\nexists$ local cell associated with the R/W shared object $X$) then
(03)    allocate local space denoted $lcell(X)$;
(04)    $x\_ptr \leftarrow PT[X]$;
(05)    $lcell(X).value \leftarrow (x\_ptr \downarrow).value$;
(06)    $lcell(X).begin \leftarrow (x\_ptr \downarrow).begin$;
(07)    $lcell(X).origin \leftarrow x\_ptr$;
(08)    $window\_bottom_T \leftarrow \max(window\_bottom_T, lcell(X).begin)$;
(09)    $lrs_T \leftarrow lrs_T \cup X$;
(10)    for each $(Y \in lrs_T)$ do $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin \downarrow).end\big)$ end for;
(11)    if $(window\_bottom_T \geq window\_top_T)$ then $\mathsf{return}(abort)$ end if
(12) end if;
(13) return $(lcell(X).value)$.
==========================================================================
operation $X.\mathsf{write}_T(v)$:
(14) if ($\nexists$ local cell associated with the R/W shared object $X$) then allocate local space $lcell(X)$ end if;
(15) $lws_T \leftarrow lws_T \cup X$;
(16) $lcell(X).value \leftarrow v$;
(17) $\mathsf{return}(ok)$.
==========================================================================
operation $\mathsf{try\_to\_commit}_T()$:
(18) lock all the objects in $lrs_T \cup lws_T$;
(19) for each $(Y \in lrs_T)$ do $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin \downarrow).end\big)$ end for;
(20) for each $(Y \in lws_T)$ do $window\_bottom_T \leftarrow \max((PT[Y] \downarrow).last\_read, window\_bottom_T)$ end for;
(21) if $(window\_bottom_T \geq window\_top_T)$ then release all locks and disallocate all local cells; $\mathsf{return}(abort)$ end if;
(22) $commit\_time_T \leftarrow$ select a (random/heuristic) time value $\in ]window\_bottom_T..window\_top_T[$;
(23) for each $(X \in lws_T)$ do $(PT[X] \downarrow).end \leftarrow commit\_time_T$ end each;
(24) for each $(X \in lws_T)$ do
(25)    allocate in shared memory a new cell for $X$ denoted $CELL(X)$;
(26)    $CELL(X).value \leftarrow lcell(X).value$; $CELL(X).last\_read \leftarrow commit\_time_T$;
(27)    $CELL(X).begin \leftarrow commit\_time_T$; $CELL(X).end \leftarrow +\infty$;
(28)    $PT[X] \leftarrow \uparrow CELL(X)$
(29) end for;
(30) for each $(X \in lrs_T)$ do
(31)    $(lcell(X).origin \downarrow).last\_read \leftarrow \max((lcell(X).origin \downarrow).last\_read, commit\_time_T)$
(32) end for;
(33) release all locks and disallocate all local cells; $last\_commit_i \leftarrow commit\_time_T$;
(34) $\mathsf{return}(commit)$.
```

Figure 2.4: Algorithm for the operations of the protocol

Lemma 3 $\forall T1, T2 \in \mathscr{C} : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_S T2$.

Proof Suppose that we have $T1 \xrightarrow{X}_{rf} T2$ ($T2$ reads the value of X written by $T1$. After the read of X by $T2$, $window\_bottom_{T2} \geq commit\_time_{T1}$ (line 08). We then have $T1 \rightarrow_S T2$. $\square_{Lemma\ 3}$

Because a transaction locks all the objects it accesses before committing (line 27), we can order totally the committed transactions that access a given object X. Let $\xrightarrow{X}_{lock}$ denote such a total order.

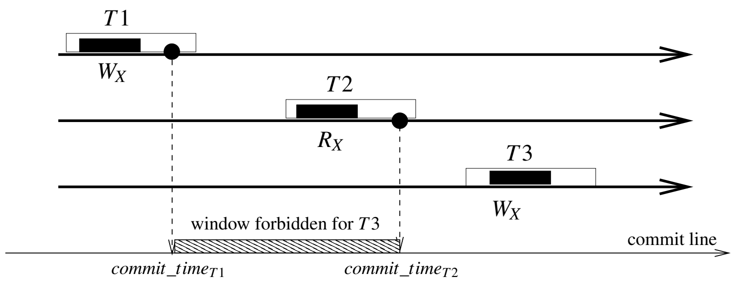


Figure 2.5: Commit intervals

```
operation try_to_commit_T():
(18)    lock all the objects in $lrs_T \cup lws_T$;
(19)    for each $(Y \in lrs_T)$ do $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin\downarrow).end\big)$ end for;
(20.A)  $commit\_set_T \leftarrow \{\ ]window\_bottom_T, window\_top_T[\ \}$;
(20.B)  for each $(X \in lws_T)$ do
(20.C)     $x\_ptr \leftarrow PT[X]$; $x\_forbid_T[X] \leftarrow \emptyset$;
(20.D)     while $((x\_ptr\downarrow).last\_read > window\_bottom_T)$ do
(20.E)          $x\_forbid_T[X] \leftarrow x\_forbid_T[X] \cup \{\ [(x\_ptr\downarrow).begin, (x\_ptr\downarrow).last\_read]\ \}$;
(20.F)          $x\_ptr \leftarrow (x\_ptr\downarrow).prev$
(20.G)     end while
(20.H)  end for;
(20.I)  $commit\_set_T \leftarrow commit\_set_T \setminus \bigcup_{X \in lws_T} \big(x\_forbid_T[X]\big)$;
(21.A)  if $(commit\_set_T = \emptyset)$ then release all locks and disallocate all local cells; return($abort$) end if;
(22.A)  $commit\_time_T \leftarrow$ select a (random/heuristic) time value $\in$ $commit\_set_T$;
(23.A)  for each $(X \in lws_T)$ do
(23.B)     $x\_ptr \leftarrow PT[X]$;
(23.C)     while $((x\_ptr\downarrow).begin > commit\_time_T)$ do $x\_ptr \leftarrow (x\_ptr\downarrow).prev$ end while;
(23.D)     $(x\_ptr\downarrow).end \leftarrow \min((x\_ptr\downarrow).end, commit\_time_T)$
(23.E)  end for;
(24.A)  for each $(X \in lws_T)$ such that $(commit\_time_T > (PT[X]\downarrow).begin)$ do
(25)            allocate in shared memory a new cell for $X$ denoted $CELL(X)$;
(26)            $CELL(X).value \leftarrow lcell(X).value$; $CELL(X).last\_read \leftarrow commit\_time_T$;
(27)            $CELL(X).begin \leftarrow commit\_time_T$; $CELL(X).end \leftarrow +\infty$;
(28.A)          $CELL(X).prev \leftarrow PT[X]$; $PT[X] \leftarrow \uparrow CELL(X)$
(29.A)  end for;
(30)    for each $(X \in lrs_T)$ do
(31)       $(lcell(X).origin\downarrow).last\_read \leftarrow \max((lcell(X).origin\downarrow).last\_read, commit\_time_T)$
(32)    end for;
(33)    release all locks and disallocate all local cells; $last\_commit_i \leftarrow commit\_time_T$;
(34)    return($commit$).
```

Figure 2.6: Algorithm for the try_to_commit() operation of the permissive protocol

Lemma 4 $X.\mathsf{write}_T() \wedge X.\mathsf{write}_{T'}() \wedge T \xrightarrow{X}_{lock} T' \Rightarrow T \rightarrow_S T'$.

Proof W.l.o.g., consider that there is no transaction T'' such that $w_{T''}(X)$ and $T \rightarrow_S T'' \rightarrow_S T'$. Because $T \xrightarrow{X}_{lock} T'$, when T' executes line 20, T has already updated $PT[x]$ and the corresponding $CELL(X)$ (because there is no T'', at this time $PT[X]\downarrow = CELL(X)$). Because $X \in lws_{T'}$, $window\_bottom_{T'} \geq (PT[X]\downarrow).last\_read \geq commit\_time_T$ (line 20). We then have $commit\_time_{T'} > commit\_time_T$ and thus $T \rightarrow_S T'$. $\Box_{Lemma\ 4}$

Corollary 1 $X.\mathsf{write}_T() \wedge X.\mathsf{write}_{T'}(X) \wedge T \rightarrow_S T' \Rightarrow T \xrightarrow{X}_{lock} T'$.

Proof The corollary follows from the fact that $\rightarrow_S$ is a total order. $\Box_{Corollary\ 1}$

Lemma 5 $\forall T1, T2 \in \mathscr{C}, \forall X : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_S T3 \rightarrow_S T2$.

Proof By way of contradiction, suppose that such a $T3$ exists. Again by way of contradiction, suppose that $T2 \xrightarrow{X}_{lock} T1$. This is not possible because $T2$ reads X before committing, and $T1$ writes X at the time of its commit (line 28). Thus $T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \xrightarrow{X}_{lock} T2$.

By Corollary 1, $X.\mathsf{write}_{T1}() \wedge X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_S T3 \Rightarrow T1 \xrightarrow{X}_{lock} T3$. We then have two possibilities: (1) $T3 \xrightarrow{X}_{lock} T2$ and (2) $T2 \xrightarrow{X}_{lock} T3$.

- Case $T3 \xrightarrow{X}_{lock} T2$. Let $lcell(X)$ be the local cell of $T2$ representing X. When $T2$ executes line 19), $T3$ has already updated the field *end* of the cell pointed by $lcell(X).origin$ with $commit\_time_{T3}$. $T2$ will then update $window\_top_{T2}$ at a smaller value than $commit\_time_{T3}$,contradicting the original assumption $T3 \rightarrow_S T2$.
- Case $T2 \xrightarrow{X}_{lock} T3$. When $T3$ executes line 20, $T2$ has already updated the field *last_read* of the cell pointed by $PT[X]$. $T3$ will then update $window\_bottom_{T3}$ at a value greater than $commit\_time_{T2}$, contradicting the original assumption $T3 \rightarrow_S T2$, which completes the proof of the lemma.

$\Box_{Lemma\ 5}$

2.12.2 Proof that the causal past of each aborted transaction is serializable

In order to show that, for each aborted transaction T, the partial order $\rightarrow_{PO}$ restricted to $past(T)$ admits a legal linear extension, we have to show that there exists a total order $\rightarrow_T$ such that the following properties hold:

1. the order $\rightarrow_T$ is a total order,
2. $\rightarrow_T$ respects the process order between transactions,
3. $\forall T1, T2 \in past(T) : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$ and,
4. $\forall T1, T2 \in past(T), \forall X : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 \in past(T) : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_T T3 \rightarrow_T T2$.

The order $\rightarrow_T$ is defined as follows:
(1) $\forall T1, T2 \in past(T) \setminus \{T\} : T1 \rightarrow_T T2$ if $T1 \rightarrow_S T2$ and,
(2) $\forall T' \in past(T) \setminus \{T\} : T' \rightarrow_T T$.

Lemma 6 *The order $\rightarrow_T$ is a total order.*

Proof The proof follows directly from the fact that $\rightarrow_T$ is defined from the total order $\rightarrow_S$ for the committed transactions in $past(T)$ (part 1 of its definition) and the fact that all these transactions are defined as preceding T (part 2 of its definition).

$\Box_{Lemma\ 6}$

Lemma 7 *The total order $\rightarrow_T$ respects the process order between transactions.*

Proof The proof follows from Lemma 2 and the definition of $\rightarrow_T$. $\Box_{Lemma\ 7}$

Lemma 8 $\forall T1, T2 \in past(T) : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$.

Proof Because no transaction can read a value from T, we necessarily have $T1 \neq T$. When $T2 \neq T$, the proof follows from the definition of $\rightarrow_T$ and Lemma 3. When $T2 = T$, the proof follows from directly from the definition of $\rightarrow_T$. $\Box_{Lemma\ 8}$

In the following lemma, we use the dual notion of the causal past of a transaction: the *causal future* of a transaction. Given a partial order $\widehat{PO}$ defined on a set of transactions, the causal future of a transaction T, denoted $future(T)$, is the set including T and all the transactions T' such that $T \rightarrow_{PO} T'$. The partial order $\widehat{PO}$ used here is the one defined in Section 2.7.1.

Lemma 9 $\forall T1, T2 \in past(T) : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 \in past(T) : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_T T3 \rightarrow_T T2$.

Proof For the same reasons as in Lemma 8, we only need to consider the case when $T2 = T$.

By way of contradiction, suppose that such a transaction $T3$ exists. Let it be the first such transaction to write X. Let $T4$ be the transaction in *future(T3)* $\cap \{T' | T' \rightarrow_{rf} T\}$ that has the biggest *commit_time* value. $T4$ is well defined because otherwise, $T3$ wouldn't be in $past(T)$. Let Y be the object written by $T4$ and read by T.

When T reads Y from $T4$, it updates $window\_bottom_T$ such that $window\_bottom_T \geq commit\_time_{T4}$ (line 08). From the fact that $T4 \in future(T3)$, we then have that $window\_bottom_T \geq commit\_time_{T3}$.

Either T reads Y from $T4$ and then reads X from $T1$, or the opposite. Let *last_op* be the latest of the two operations. During *last_op*, T updates $window\_bottom_T$. Due to the fact that $T3 \in past(T)$, $T3$ has already updated the pointer $PT[Z]$ for some object Z (line 28), and thus has already updated the field *end* (line 23) of the cell pointed by $lcell(X).origin$ ($lcell(X)$ being the local cell of T representing X). T will then observe $window\_bottom_T \geq window\_top_T$ (line 11) and will not complete *last_op*, again a contradiction, which completes the proof of the lemma. $\Box_{Lemma\ 9}$

2.12.3 VWC and read invisibility

Theorem 2 *The algorithm presented in Figure 2.4 satisfies virtual world consistency and implements invisible read operations*

Proof The proof that the algorithm presented in Figure 2.4 satisfies virtual world consistency follows from Lemmas 1, 2, 3, 5, 6, 7, 8 and 9.

The fact that, for any shared object X and any transaction T, the operation $X.\mathsf{read}_T()$ is invisible follows from a simple examination of the text of the algorithm implementing that operation (lines 01-13 of Figure 2.4): there is no write into the shared memory. $\Box_{Theorem\ 2}$

2.13 Improving the base protocol described in Figure 2.4

This section considers three improvements of the base protocol previously presented and proved. Those concern the following points: how the useless cells can be collected, how read operations can be made fast and how serializability can be replaced by linearizability.

2.13.1 Garbage collecting useless cells

This section presents a relatively simple mechanism that allows shared memory cells that have become inaccessible to be collected for recycling. This mechanism is based on the pointers *next*, two additional shared arrays, the addition of new statements to both $X.\mathsf{read}_T()$ and the $\mathsf{try\_to\_commit}_T()$, and a background task BT.

Additional arrays The first is an array of atomic variables denoted $\mathit{LAST\_COMMIT}[1..m]$ (remember that m is the number of sacred objects). This array is such that $\mathit{LAST\_COMMIT}[X]$ (which is initialed to 0) contains the date of the last committed transaction that has written into X. Hence, the statement "$\mathit{LAST\_COMMIT}[X] \leftarrow \mathit{commit\_time}_T$" is added in the **do** ... **end** part of line 23.

The second array, denoted $\mathit{MIN\_READ}[1..n]$, is made up of one-writer/one-reader atomic registers (let us recall that n is the total number of processes). $\mathit{MIN\_READ}[i]$ is written by p_i only and read by the background task BT only. It is initialized to $+\infty$ and reset to its initial value value when p_i terminates $\mathsf{try\_to\_commit}_T()$ (i.e., just before returning at line 21 or line 34 of Figure 2.4). When $\mathit{MIN\_READ}[i] \neq +\infty$, its value is the smallest commit date of a value read by the transaction T currently executed by p_i. Moreover, the following statement has to be added after line 03 of the $X.\mathsf{read}_T()$ operation:

$$\mathit{MIN\_READ}[i] \leftarrow \min(\mathit{MIN\_READ}[i], \mathit{LAST\_COMMIT}[X]).$$

Managing the *next* pointers When a process executes the operation $\mathsf{try\_to\_commit}_T()$ and commits the corresponding transaction T, it has to update a pointer *next* in order to establish a correct linking of the cells implementing X. To that end, p_i has to execute $(PT[X]\downarrow).\mathit{next} \leftarrow\uparrow \mathit{CELL}[X]$ just before updating $PT[X]$ at line 28.

The background task BT This sequential task, denoted BT, is described in Figure 2.7. It uses a local array denoted $last_valid\_pt[1..m]$ such that $\mathit{last\_valid\_pt}[X]$ is a pointer initialized to $PT[X]$ (line 01). Then its value is a pointer to the cell containing the oldest value of X that is currently accessed by a transaction (this is actually a conservative value).

The body of task BT is an infinite loop (lines 02-12). BT first computes the smallest commit date still useful (line 03). Then, for every shared object X, BT scans the list from $\mathit{last\_valid\_pt}[X]$ and releases the space occupied by all the cells containing values of X that are no longer accessible (lines 06-09), after which it updates $\mathit{last\_valid\_pt}[X]$ to its new pointer value (line 10). Lines 06 and 07 uses two consecutive *next* pointers. Those are due to the maximal concurrency allowed by the algorithm, more specifically, they prevent an $X.\mathsf{read}_T()$ operation from accessing a released cell.

It is worth noticing that the STM system and task BT can run concurrently without mutual exclusion. Hence, BT allows for maximal concurrency. The reader can also observe that such a maximal concurrency has a price, namely (as seen in line 06 where the last two cells with

```
(01) init: for every $X$ do $last\_valid\_pt[X] \leftarrow PT[X]$ end for.

background task $BT$:
(02) repeat forever
(03)     $min\_useful \leftarrow \min(\{MIN\_READ[i]\}_{1\leq i\leq n})$;
(04)     for every $X$ do
(05)         $last \leftarrow last\_valid\_pt[X]$;
(06)         while $\big((last \neq PT[X]) \wedge (last\downarrow).next\downarrow).next \neq \bot)$
(07)                 $\wedge[(((last\downarrow).next\downarrow).next\downarrow).commit\_time < min\_useful]\big)$
(08)           do $temp \leftarrow last$; $last \leftarrow (last\downarrow).next$; release the cell pointed to by $temp$
(09)         end while;
(10)         $last\_valid\_pt[X] \leftarrow last$
(11)     end for
(12) end repeat.
```

Figure 2.7: The cleaning background task *BT*

commit time smaller than *min_useful* are kept) for any shared object X, task *BT* allows all -but at most one- useless cells to be released.

2.13.2 Expediting read operations

Let us consider the invocations $X.\mathsf{read}_T()$ (those are issued by T). From the second one, none of these invocations access the shared memory. As described in Figure 2.4, the first invocation $X.\mathsf{read}_T()$ entails the execution of line 10 whose costs is $O(|lrs_T|)$. Interestingly, it is possible to define "favorable circumstances" in which the execution of this line can be saved when $X.\mathsf{read}_T()$ is invoked for the first time by T. Its cost becomes then $O(1)$. To that end, each process p_i is required to manage an additional local variable denoted $earliest\_read_T$ (that is initialized to $+\infty$ at line 01 of Figure 2.4). Line 10 of $X.\mathsf{read}_T()$ is then replaced by:

```
if $(lcell(X).commit\_time > earliest\_read_T)$
    then Code of line 10 else $earliest\_read_T \leftarrow lcell(X).commit\_time$ end if.
```

Hence, the protocol allows for *fast* read operations in favorable circumstances. (It is even possible, at the price of another additional control variable, to refine the predicate used in the previous statement in order to increase the number of favorable cases. Such an improvement is described in Appendix 2.15. It is important to notice that, while these *fast* read operations are possible when the consistency condition is VWC, they are not when it is opacity.)

2.13.3 From serializability to linearizability

The IR_VWC_P protocol guarantees that the committed transactions are serializable. A simple modification of the protocol allows it to ensures the stronger "linearizability" condition [135] instead of the weaker "serializability"condition. The modification assumes a common global clock that processes can read by invoking the operation System.get_time(). It is as follows.

- The statement $window\_bottom_T \leftarrow last\_commit_i$ at line 01 of $\mathsf{begin}_T()$ is replaced by the statement $window\_bottom_T \leftarrow$ System.get_time().

- The following statement is added just between line 19 and line 20 of $\mathsf{try\_to\_commit}_T()$ (Figure 2.4):

  if $(window\_top_T = +\infty)$ **then** $window\_top_T \leftarrow$ $\mathsf{System.get\_time}()$ **end if**.

It is easy to see that these modifications force the commit time of a transaction to lie between its starting time and its end time. Let us observe that now the disjoint access parallelism property remains to be satisfied but for the accesses to the common clock.

2.14 Proof of the probabilistic permissiveness property

In order to show that the protocol is probabilistically permissive with respect to virtual world consistency, we have to show the following. Given a transaction history that contains only committed transactions, if the partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ accepts a legal linear extension (as defined in Section 2.7), then the history is accepted (no operation returns abort) with positive probability. As in [38], we consider that operations are executed in isolation. It is important to notice here that only operations, not transactions, are isolated. Different transactions can still be interlaced.

Let $\rightarrow_S$ be the order on transactions defined by the protocol according to the $commit\_time_T$ variable of each transaction T (this order has already been defined in Section 2.12).

Lemma 10 *Let T and T' be two committed transactions such that $T \not\rightarrow_{PO} T'$ and $T' \not\rightarrow_{PO} T$. If there is a legal linear extension of $\widehat{PO}$ in which T precedes T', then there is a positive probability that $T \rightarrow_S T'$.*

Proof Because $T \not\rightarrow_{PO} T'$, the set $past(T) \backslash past(T')$ is not empty ($past(T)$ does not contain T'). Let $biggest\_ct_{T,T'}$ be the biggest value of the $commit\_time$ variables (chosen at line 22.A) of the transactions in the set $past(T) \cap past(T')$ if it is not empty, or 0 otherwise. Suppose that every transaction in $past(T) \backslash past(T')$ chooses the smallest value possible for its $commit\_time$ variable. These values cannot be constrained (for their lower bound) by a value bigger than $biggest\_ct_{T,T'}$, thus they can all be smaller than $biggest\_ct_{T,T'} + \varepsilon$ for any given ε.

Suppose now that T' chooses a value bigger than $biggest\_ct_{T,T'} + \varepsilon$ for its $commit\_time$ variable. This is possible because, for a given transaction $T1$, the upper bound on the value of $commit\_time_{T1}$ can only be fixed by a transaction $T2$ that overwrites a value read by $T1$ (lines 10 and 19). Suppose now that $T1$ is T'. If there was such a transaction $T2$ in $past(T)$, then there would be no legal linear extension of $\rightarrow_{PO}$ in which T precedes T'. Thus if there is a legal linear extension of $\rightarrow_{PO}$ in which T precedes T', then there is a positive probability that $T \rightarrow_S T'$. $\Box_{Lemma\ 10}$

Lemma 11 *Let $\widehat{PO} = (PO, \rightarrow_{PO})$ be a partial order that accepts a legal linear extension. Every operation of each transaction in PO does not return abort with positive probability.*

Proof $X.\mathsf{write}_T()$ operations cannot return *abort*. Therefore, we will only consider the operations $X.\mathsf{read}_T()$ and $\mathsf{try\_to\_commit}_T()$.

Let $\rightarrow_{legal}$ be a legal linear extension of $\rightarrow_{PO}$. Let op be an operation executed by a transaction. Let $\mathscr{C}_{op}$ be the set of transactions that have ended their $\mathsf{try\_to\_commit}()$ operation before operation op is executed (because we consider that the operations are executed in isolation, this set is well defined). Let $\rightarrow_{op}$ be the total order on these transactions defined by the protocol.

From Lemma 10, two transactions that are not causally related can be totally ordered in any way that allows a legal linear extension of $\rightarrow_{PO}$. There is then a positive probability that $\rightarrow_{op} \subset \rightarrow_{legal}$. Suppose that it is true. Let T be the transaction executing op. Let $T1$ and $T2$ be the transactions directly preceding and following T in $\rightarrow_{legal}$ restricted to $\mathscr{C}_{op} \cup \{T\}$ if they exist. If $T1$ does not exist, then $window\_bottom_T = 0$ at the time of all the operation of T, and thus T can execute successfully all its operations. Similarly, if $T2$ does not exist, then $window\_top_T = +\infty$ at the time of all the operation of T, and thus T can execute successfully all its operations. We will then consider that both $T1$ and $T2$ exist.

Because $\rightarrow_{legal}$ is a legal linear extension of $\rightarrow_{PO}$, any transaction from which T reads a value is either $T1$ or a transaction preceding $T1$ in $\rightarrow_{op}$ (line 08) resulting in a value for $window\_bottom_T$ that is at most $commit\_time_{T1}$. Similarly, any transaction that overwrote a value read by T at the time of op is either $T2$ or a transaction following $T2$ in $\rightarrow_{op}$ (line 10) resulting in a value for $window\_top_T$ that is at least $commit\_time_{T2}$. All the read operations of T will then succeed (line 11).

Let us now consider the case of the try_to_commit() operation. Because all read operations have succeeded, the set $commit\_set_T =]window\_bottom_T..window\_top_T[$ (line 20.A) is not empty and must contain the set $]commit\_time_{T1}..commit\_time_{T2}[$. Because $\rightarrow_{legal}$ is a legal linear extension of $\rightarrow_{PO}$, if T writes to an object X then T cannot be placed between two transactions $T3$ and $T4$ such that $T3$ reads a value of object X written by $T4$.

Because these intervals (represented by $x\_forbid_T[X]$, lines 20.B to 20.H) are the only ones removed from $commit\_set_T$ (line 20.I) and because there is a legal linear extension of $\rightarrow_{PO}$ which includes T, $commit\_set_T$ is not empty and the transaction can commit successfully, which ends the proof of the lemma. $\Box_{Lemma\ 11}$

Theorem 3 *The algorithm presented in Figure 2.4, where the* try_to_commit() *operation has been replaced by the one presented in Figure 2.6, is probabilistically permissive with respect to virtual world consistency.*

Proof From Lemma 11, all transactions of a history can commit with positive probability if the history is virtual world consistent, which proves the theorem. $\Box_{Theorem\ 3}$

2.15 More on fast read operations

Virtual world consistency vs opacity Unlike opacity, a live transaction satisfying the VWC consistency criterion only has to be concerned with its causal past in order not to violate consistency. When a new transaction commits in an opaque STM a live transaction has always to consider this transaction. In contrast in a VWC STM, a live transaction needs to consider only it if it is in the live transaction's causal past.

We can take advantage of this observation in order to increase the number of times a transaction performs fast reads. This is not without a trade off though, allowing a transaction to only consider its causal past means that in certain cases transactions with no realtime-compliant legal linear extension with previously committed transactions will have their abort operation delayed. Or in other words a transaction that is doomed to abort could possibly be allowed to stay alive longer. In our STM we also have the cost of using an additional control variable. How much efficiency is gained or lost by this will certainly depend on the execution.

The method described below is not the only way to take advantage of VWC for fast reads. It has to be seen as one among several possible enhancements. The idea is that when a live transaction T reads a value written by some other transaction $T_1 \notin past(T)$,then T_1's causal past is added to $past(T)$. But if the commit time of the transaction in $past(T_1)$ with the largest commit time is smaller then the commit time of all transactions that T has read from then it is impossible for any of the transactions in $past(T_1)$ to overwrite any of T's reads. In this case only the transaction T_1 itself could overwrite a value read by T causing T to not be VWC, but this is only possible if T_1 has overwritten a value that was previously written by a transaction with commit time later then the commit time of the earliest transaction that T has read from. Thus using some extra control variables allows us to perform fast read operations in these cases.

An implementation This part discusses an implementation of the previous idea. While a transaction is live it keeps a local variable called $latest\_read_T$ initialized as $commit\_time_i$ and updated during each $X.read_T()$ operation to the largest commit time of the transactions it has read from so far. When T executes line 20 of the try_to_commit$_T()$ operation, the variable $latest\_read_T$ is (possibly) increased to the largest commit time of the transaction for the values T is overwriting. A boolean control variable, $overwrites\_latest\_read_T$ (initialized to *false*) is set to *true* if $latest\_read_T$ is modified. To that end line20 is replaced by the code described in Figure 2.8.

```
(13) for each $(Y \in lws_T)$ do
(14)     $window\_bottom_T \leftarrow \max((PT[Y]\downarrow).last\_read, window\_bottom_T)$;
(15)     if $((PT[Y]\downarrow).commit\_time \geq latest\_read_T)$ then
(16)         $latest\_read_T \leftarrow (PT[Y]\downarrow).commit\_time$; $overwrites\_latest\_read_T \leftarrow true$
(17)     end if
(18) end for.
```

Figure 2.8: Fast read: Code to replace line 20 of Figure 2.4

Moreover, when a transaction commits, the values of $latest\_read_T$ and $overwrites\_latest\_read_T$ have now to be stored in shared memory along with the other values in $CELL(X)$ for each variable X written by T.

Finally, line 10 of X.read$_T()$ is replaced by the following statement:

```
(a)   if $\big((CELL(X).latest\_read > earliest\_read_T) \vee$
(b)       $(CELL(X).latest\_read = earliest\_read_T \wedge CELL(X).overwrites\_latest\_read)\big)$
          then Code of line 10 end if;
      $earliest\_read_T \leftarrow \min(lcell(X).commit\_time, earliest\_read_T)$;
      $latest\_read_T \leftarrow \max(lcell(X).commit\_time, latest\_read_T)$.
```

Discussion It can be seen that there are two cases when update_window_top$_T()$ will be required to be executed.

- The first corresponds to line (a), i.e., when the predicate $CELL(X).latest\_read > earliest\_read_T$ (let T_1 be the transaction that wrote $CELL(X)$) is satisfied, meaning that there is at least one transaction in the causal past of T that has a commit time earlier than either a transaction in the causal past of T_1, or a transaction that has written a value overwritten by T_1. Thus a value read by T might have been overwritten by T_1 or $past(T_1)$ and update_window_top$_T()$ needs to be executed.

- The second corresponds to line (b), i.e., when the predicate $CELL(X).latest\_read = earliest\_read_T \wedge cell(X).overwrites\_latest\_read$ is true. First, as each transaction has a unique commit time, when $CELL(X).latest\_read = earliest\_read_T$ then the transactions described by $CELL(X).latest\_read$ and $earliest\_read_T$ are actually the same transaction, call this transaction T_2. So if the boolean variable $CELL(X).overwrites\_latest\_read$ is false, then T_1 just reads a value from T_2 resulting in there being no possibility of a read of T being overwritten. Otherwise if $CELL(X).overwrites\_latest\_read$ is satisfied, then T_1 is overwritten a value written by T_2 and update_window_top$_T()$ needs to be executed. In all other cases a fast read is performed.

Fast read operations and opacity As mentioned previously performing fast reads in this way is only possible in VWC. We give below a counterexample (Figure 2.9) in order to show that they are not compatible with opacity.

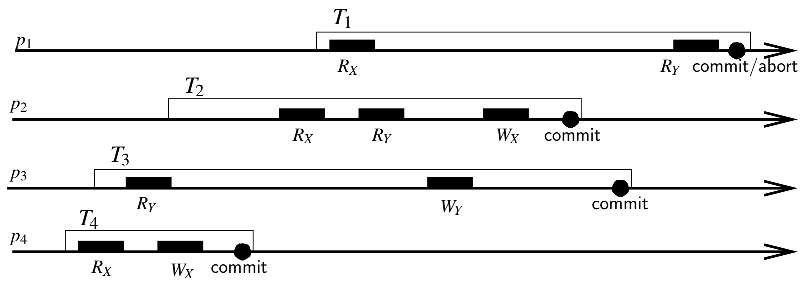


Figure 2.9: Example of fast reads that would violate opacity

In this figure we have T_4 committing first, then T_2 commits and must be serialized after T_4 because it reads the value of X written by T_4. Next T_3 commits and must be serialized after T_2 because it overwrites the value of Y read by T_2. Finally we have T_1 which first reads the value of X written by T_4 then reads the value of Y written by T_3. Now T_1 violates opacity because it must be serialized before T_2 (T_2 overwrites the value of X it read) and after T_3 (it reads the value of Y written by T_3), but we already know that T_2 is serialized before T_3. On the other hand, T_1 is VWC because its causal past does not contain T_2. Now we just need to show that update_window_top$_T()$ is not executed during the execution of Y.read$_{T_1}()$. After T_1 performs X.read$_{T_1}()$ the variable $earliest\_read_{T_1}$ is set to the commit time of T_4. The value $latest\_read_{T_3}$ is the commit time of transaction T_0 so when T_3 commits we have $CELL((Y).latest\_read$ also set to this value. During Y.read$_{T_1}()$ we have $CELL(Y).latest\_read < earliest\_read_{T_1}$ and update_window_top$_{T_1}()$ is not executed.

2.16 Making read operations invisible at commit time

Read invisibility vs try_to_commit$_T()$ **invisibility** Let us observe that read invisibility requires that read operations be invisible when they are issued by a transaction T, but does not require they remain invisible at commit time. This means that the shared memory is not modified during the read operation, but a try_to_commit$_T()$ operation is allowed to modify shared memory locations associated with base objects read by transaction T. Interestingly though this does not always need to be the case.

Similarly to fast read operations where line 10 of X.read$_T()$ does not need to be always executed, a read does not always need to be made visible during the try_to_commit$_T()$ operation. There are two locations in try_to_commit$_T()$ that modify shared memory with respect to objects $X \in lrs_T \setminus lws_T$. The first is on line 31 where the value of $cell(X).last\_read$ is updated for each object $X \in lrs_T$. The second is on line 18 where each object $X \in lrs_T$ is locked. Concerning efficiency and scalability, not only can locking and writing to shared memory be considered expensive operations, but is is desirable for reads to be completely invisible.

Discussion: When write to shared memory and lock can be avoided Assume some shared variable Y read by transaction T. First consider the write to $cell(Y).last\_read$. In the algorithm, the only time the *last_read* field of a cell is read is on line 20 of the try_to_commit$_T()$ operation. Since *last_read* is only read for objects in lws_T, and T locks all objects in lws_T, T is guaranteed to be accessing the most recent cell. This means that, during a try_to_commit$_T()$ operation on line 31, if $lcell(Y).origin$ is not the latest cell for Y, then the value of *last_read* written will never be accessed and there is no reason to write the value.

So the write on line 31 is not necessary when $lcell(Y).origin$ is not the latest cell, what about the lock associated with Y? When is it not necessary to lock objects in lrs_T? This turns out to not be necessary in the same case. If line 31 is not executed for some variable $Y \in lrs_T$, then the only other place a cell of Y is accessed in the try_to_commit$_T()$ operation is on line 19. Here the only shared memory accessed is $(lcell(Y).origin \downarrow).end$. From the construction of the algorithm $(lcell(Y).origin \downarrow).end$ will be updated at most once, therefore if the loop iteration for Y on line 10 of X.read$_T()$ (where X and Y may or may not be the same variable) had been executed previously with $(lcell(Y).origin \downarrow).end \neq +\infty$ (meaning $(lcell(Y).origin \downarrow).end$ had been updated previously), then the loop iteration for Y does not need to be executed again which results in Y not needing to be locked.

An implementation It follows from the previous discussion that a read operation of a shared variable Y can be made invisible at commit time if $(lcell(Y).origin \downarrow).end$ had not equal to $+\infty$ during the loop iteration for Y on line 10 for any execution of X.read$_T()$. Implementing this in the algorithm becomes easy, line 10 must be replaced by the following.

```
for each $(Y \in lrs_T)$ do
    $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin \downarrow).end\big)$;
    if $((lcell(Y).origin \downarrow).end \neq +\infty)$ then $lrs_T \leftarrow lrs_T \setminus \{Y\}$ end if
end for.
```

Using this improvement there is a possibility for a transaction T (that performs at least one read) to have some or all of its reads to be invisible during the try_to_commit$_T()$ operation. Consequently a read only transaction has a possibility to be completely invisible at commit time, meaning that the try_to_commit$_T()$ operation will just immediately return *commit* without doing any work.

Additional Benefits It is also worth noting that this improvement can also improve the performance of the read operations performed by the STM. This is because when the one of the new lines added is executed an object is removed from lrs_T, and the cost of the read operation depends on the size of lrs_T. In the best case this can cause the cost of a read operation to be $O(1)$ instead of $O(|lrs_T|)$ (note that this works concurrently along with fast reads as described in Appendix 2.15).

2.17 Substraction on sets of intervals (line 20.H of Figure 2.6)

The subtraction operation on sets of intervals of real numbers $commit\_set_T \backslash x\_forbid_T[X]$ has the usual meaning, which is explained with an example in Figure 2.10.

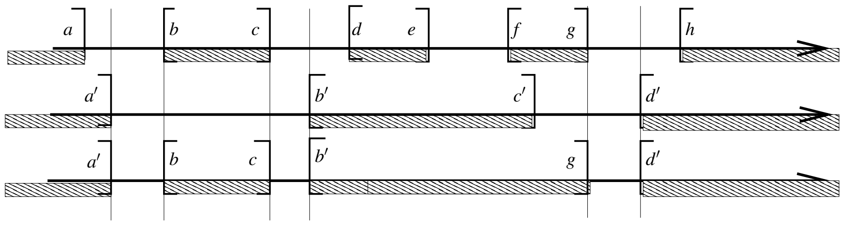


Figure 2.10: Subtraction on sets of intervals

The top line represents the value of $commit\_time_T$ that is made up of 4 intervals, $commit\_time_T = \{\ ]a..b[,\ ]c..d[,\ ]e..f[,\ ]g..h[\ \}$. The black intervals denote the time intervals in which T cannot be committed. The set $x\_forbid_T[X]$ is the set of intervals in which T cannot commit due to the access to X issued by T and other transactions. This set is depicted in the second line of the where we have $x\_forbid_T[X] = \{\ [0..a'],\ [b'..c'], [d'..+\infty[\ \}$. The last line of the figure, show that we have $commit\_time_T \backslash x\_forbid_T[X] = \{\ ]a'..b[,\ ]c..b'[,\ ]g..d'[\ \}$.

2.18 About the predicate of line 24.A of Figure 2.6

This section explains the meaning of the predicate used at line 24.A: $commit\_time_T > (PT[X] \downarrow).begin$. This predicate controls the physical write in a shared memory cell of the value v that T wants to write into X. It states that the value is written only if $commit\_time_T > (PT[X] \downarrow).begin$. This is due to the following reason.

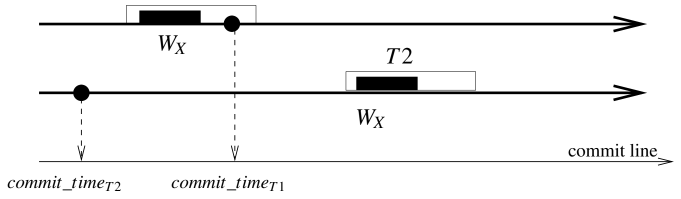


Figure 2.11: Predicate of line 24.A of Figure 2.6

Let us remember that a transaction is serialized at a random point that belongs to its current set of intervals $current\_set_T$. Moreover as we are looking for serializable transactions, the serialization points of two transactions $T1$ and $T2$ are not necessarily real time-compliant, they depend only of their sets $current\_set_{T1}$ and $current\_set_{T1}$, respectively.

An example is described in Figure 2.11. Transaction $T1$, that invokes $X.\mathsf{write}_{T1}()$, executes first (in real time), commits and is serialized at (logical) time $commit\_time_{T1}$ as indicated on the Figure. Then (according to real time) transaction $T2$, that invokes also $X.\mathsf{write}_{T2}()$, is invoked and then commits. Moreover, its commit set and the random selection of its commit time are such that $commit\_time_{T2} < commit\_time_{T1}$. It follows that $T2$ is serialized before $T1$. Consequently, the last value of X (according to commit times) is the one written by $T1$, that has overwritten the one written by $T2$. The predicate $commit\_time_T > (PT[X] \downarrow).begin$ prevents the committed transaction $T2$ to write its value into X in order write and read operations on X issued by other transactions be in agreement with the serialization order defined by commit times.

Chapter 3

Universal Constructions and Transactional Memory

3.1 Universal Constructions

3.2 Aborts, Commits, and Liveness in Transactional Memory

3.3 A Universal Construction for Transaction Based Multi-Process Programs

3.3.1 Proof of Linearizability

3.3.2 Proof of Liveness

3.3.3 Computation Complexity

3.4 ***An Implementation***

3.5 Introduction

Lock-based concurrent programming A *concurrent object* is an object that can be concurrently accessed by different processes of a multiprocess program. It is well known that the design of a concurrent program is not an easy task. To that end, base synchronization objects have been defined to help the programmer solve concurrency and process cooperation issues. A major step in that direction has been (more than forty years ago!) the concept of *mutual exclusion* [31] that has given rise to the notion of a *lock* object. Such an object provides the processes with two operations (lock and unlock) that allows a single process at a time to access a concurrent object. Hence, from a concurrent object point of view, the lock associated with an object allows transforming concurrent accesses on that object into sequential accesses. Interestingly, all the books on synchronization and operating systems have chapters on lock-based synchronization. In addition, according to the abstraction level supplied to the programmer, a lock may be encapsulated into a linguistic construct such as a *monitor* [50] or a *serializer* [49].

Unfortunately locks have drawbacks. One is related to the granularity of the object protected by a lock. More precisely, if several data items are encapsulated in a single concurrent object, the inherent parallelism the object can provide can be drastically reduced. This is for example the case of a queue object for which concurrent executions of enqueue and dequeue operations should be possible as long as they are not on the same item. Of course a solution could consist of considering each item of the queue as a concurrent object, but in that case, the operations enqueue and dequeue can become very difficult to design and implement. More severe drawbacks associated with locks lie in the fact that lock-based operations are deadlock-prone and cannot be easily composed.

Hence the question: how to ease the job of the programmer of concurrent applications? A (partial) solution consists of providing her/him with an appropriate library where (s)he can find correct and efficient implementations of the most popular concurrent data structures (e.g., [110, 53]). Albeit very attractive, this approach does not solve entirely the problem as it does not allow the programmer to define specific concurrent objects that take into account her/his particular synchronization issues.

The Software Transactional Memory approach The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory was first proposed (nearly twenty years ago!) by Herlihy and Moss to implement concurrent data structures [78]. It has then been implemented in software by Shavit and Touitou [57] and has recently gained great momentum as a promising alternative to locks in concurrent programming [36, 42, 51, 54]. Interestingly enough, it is important to also observe that the recent advent of multicore architectures has given rise to what is called the *multicore revolution* [44] that has rang the revival of concurrent programming.

Transactional memory abstracts away the complexity associated with concurrent programming by replacing locking with atomic execution units. In that way, the programmer has to focus on where atomicity is required and not on the way it must be realized. The aim of an STM system is consequently to discharge the programmer from the direct management of the synchronization that is entailed by accesses to concurrent objects.

More generally, STM is a middleware approach that provides the programmers with the *transaction* concept (this concept is close but different from the notion of transactions encountered in database systems [36]). A process is designed as (or decomposed into) a sequence of transactions, with each transaction being a piece of code that, while accessing concurrent objects, always appears as if it was executed atomically[1]. The job of the programmer is only to state which units of computation have to be atomic. He does not have to worry about the fact that the objects accessed by a transaction can be concurrently accessed. The programmer is not concerned by synchronization except when (s)he defines the beginning and the end of a transaction. It is then the job of the STM system to ensure that transactions are executed as if they were atomic.

Let us observe that the "spirit/design philosophy" that has given rise to STM systems is not new: it is related to the notion of *abstraction level*. More precisely, the aim is to allow the programmer to focus and concentrate only on the problem (s)he has to solve and not on the base machinery needed to solve it. As we can see, this is the approach that has replaced assembly languages by high level languages and programmer-defined garbage collection by automatic garbage collection. STM can be seen as a new concept that takes up this challenge when

[1]Actually, while the word "*transaction*" has historical roots, it seems that "*atomic procedure*" would be more appropriate because "transactions" of STM systems are computer science objects that are different from database transactions. We nevertheless continue using the word "*transaction*" for historical reasons.

considering synchronization issues.

The state of affairs and related works Of course, a solution in which a single transaction is executed at a time trivially implements transaction atomicity, but is inefficient on a multiprocessor system (as it allows only non-transactional code to execute in parallel). Instead, an STM system has to allow several transactions to execute concurrently, but then it is possible that they access the same concurrent objects in a conflicting manner. In that case some of these transactions might have to be aborted. Hence, in a classical STM system there is an *abort/commit* notion associated with transactions.

Several approaches have been proposed to cope with aborted transactions. In some systems the management of aborted transactions is left to the application programmer (similarly to exception handling encountered in some systems). Other systems offer "best effort semantics" (which means that there is no provable strong guarantee), an interesting example of this approach is the technique called *steal-on-abort* [26]. Its base principle is the following one. If a transaction $T1$ is aborted due to a conflict with a transaction $T2$, $T1$ is assigned to the processor that executed $T2$ in order to prevent a new conflict between $T1$ and $T2$.

Another approach consists in designing schedulers that perform particularly well in appropriate workloads, for example the case of read-dominated workloads is deeply investigated in [27]. Yet another approach consists of enriching the system with a contention manager whose aim is to improve performance and ensure (best effort or provable) progress guarantees. An associated theory is described in [39]. Failure detector-based contention managers (and corresponding lower bounds) are described in [40]. A construction to execute parallel programs made up of *atomic blocks* that have to be dispatched to queues accessed by threads (logical processors) is presented in [58]. Other related works are presented in Section 3.6.

Aim of the paper The previous observations show that, contrarily to the initial hope, STM systems proposed so far either do not entirely free the programmer from the management of synchronization-related issues or require specific adaptation of both the compiler and the underlying scheduler. (Actually, the management of aborted transactions can be seen as the transactional counterpart of the deadlock management techniques - prevention/resolution - encountered in lock-based systems.) Ideally, an STM system should be such that, from the programmer point of view, each transaction invoked by a process is executed exactly once. Said differently, this means that, at the programming level, there should be no notion of abort/commit associated with a transaction. Hence, the programmer is concerned neither by the way synchronization is implemented, nor by the way transactions are executed[2].

This paper is a step in that direction. From a conceptual point of view, it advocates that the abstraction level offered to the programmer and the technicalities needed at the implementation level have to be totally dissociated[3]: the programmer does not need to be aware of the way the underlying STM system works (i.e., similarly to the fact that deadlock management and parameter passing mechanisms associated with procedure calls are hidden to the programmer,

[2]It is nevertheless important to notice that while a transaction can access local variables and concurrent objects, we assume here that it does not issue inputs/outputs. Those are done in the non-transactional code of the corresponding process. Hence a transaction is somewhat restricted in what it can do.

[3]It is important to notice that such an approach is adopted in some systems where the abort/commit notion is unknown to the programmer and the pair "compiler + scheduler" is designed in such a way that every transaction is "practically always" eventually executed [34].

abort/commit is an implementation-related notion that has to remain confined to the transaction implementation level).

Content of the paper The paper addresses the previous issue by presenting a construction (STM system) where, at the programming level, every transaction invocation is executed exactly once. The notion of abort/commit is relegated to the implementation and not known to the programmer. Then, as previously indicated, the job of a programmer is to write her/his concurrent program in terms of cooperating sequential processes, each process being made up of a sequence of transactions (plus possibly some non-transactional code). At the programming level, any transaction invoked by a process is executed exactly once (similarly to a procedure invocation in sequential computing). Moreover, from a global point of view, any execution of the concurrent program is linearizable [135], meaning that all the transactions appear as if they have been executed one after the other in an order compatible with their real-time occurrence order. Hence, from the programmer point of view, the progress condition associated with an execution is a very classical one, namely, *starvation-freedom.*

The proposed construction can be seen as a *universal construction* for transaction-based concurrent programs. In addition to providing such a construction, an important aim of the paper is to investigate the design principles this construction relies on (efficiency issues are not part of our study). Interestingly, the fact that there is no abort/commit notion at the programming level and the very existence of this construction constitute an interesting feature which feeds the debate on the "liveness" of STM systems. Moreover, the proposed construction has a noteworthy feature: it is lock-free (in the sense it uses no lock). It also ensures linearizable X-ability. X-ability, or exactly-once ability, was originally defined by Frølund and Guerraoui in [37] as a correctness condition for replicated services such as primary-backup. In this model there are actions, such as transactions, that cause some side effect. For a service to satisfy X-ability every invoked action and its side effect must be observed as if it had happened exactly once. In order to ensure this, actions might be executed multiple times by the underlying system. Given this requirement, x-ability concerns both correctness and liveness, and can compliment concurrency correctness conditions.

In order to build such a construction, the paper assumes an underlying multiprocessor where the processors communicate through a shared memory that provides them with atomic read/write registers, compare&swap registers and fetch&increment registers.

As we will see, the underlying multiprocessor system consists of m processors where each processor in charge of a subset of processes. The multiprocess program, defined by the programmer, is made up of n processes where each process is a separate thread of execution. We say that a processor *owns* the corresponding processes in the sense that it has the responsibility of their individual progress. Given that at the implementation level a transaction may abort, the processor P_x owning the corresponding process p_i can require the help of the other processors in order for the transaction to be eventually committed. The implementation of this helping mechanism is at the core of the construction (similarly to the helping mechanism used to implement wait-free operations despite any number of process crashes [43]). As we will see, the main technical difficulties lie in ensuring that (1) the helping mechanism allows a transaction to be committed exactly once and (2) each processor P_x ensures the individual progress of each process p_i that it owns. As we can see, from a global point of view, the m processors have to cooperate in order to ensure a correct execution/simulation of the n processes.

Roadmap The paper is made up of 8.9 sections. Section 3.6 presents related works. Section 3.7 presents the model offered to the programmer and the underlying multiprocessor model on which the STM system is built. Section 3.8 presents the universal construction (STM system) for transaction-based concurrent programs. Section 6.11 addresses correctness. Section 3.10 shows that the number of times a transaction can be executed before being committed is bounded. Finally, Section 8.9 concludes the paper by discussing additional features of the proposed construction.

3.6 Related work

As mentioned previously, the paper presents a new STM construction that ensures that every transaction issued by a process is necessarily committed and each process makes progress. To our knowledge, this is the first STM system we know of to combine these concepts, but much work has been done previously on these two "liveness"-related concepts when implemented separately. This section gives a short overview of that work. The presentation is not entirely fair as the major part of the papers that are cited here are efficiency-oriented while our construction is more theory-oriented.

Contention Management The idea is here to keep track of conflicts between transactions and have a separate entity, usually called *contention manager*, decide what action to take (if any). Some of these actions include aborting one or both of the conflicting transactions, stalling one of the transactions, or doing nothing. The idea was first (as far as we know) proposed in the dynamic STM system (called DSTM) [80] and much research has been done on the topic since then.

An overview of different contention managers and their performance is presented in [39]. Interestingly the authors find that there is no "best" contention manager and that the performance depends on the application. The notion of a *greedy contention manager* they propose ensures that every issued transaction eventually commits. This is done by giving each transaction a time-stamp when it is first issued and, once the time-stamp reaches a certain age, the system ensures that no other transaction can commit that will cause this transaction to abort. Similarly to the "Wait/Die" or "Wound/Wait" strategies used to solve deadlocks in some database systems [55], preventing transactions from committing is achieved by either aborting them or directing them to wait. By doing this it is obvious that processes with conflicting transactions make progress. Their progress depends on the process that issued the transaction with the oldest time-stamp to commit.

As discussed in this paper, committing every issued transaction should be ensured by an STM. Unfortunately, in order to ensure these properties, the greedy contention manager is considered by many to be too expensive to always use and still provide good performance. To get around this some modern STM's (e.g., for example TinySTM [73] or SwissSTM [32]) use less expensive contention management until a transaction has been aborted a certain number of times at which point greedy contention management is used.

Transactional Scheduling As we have seen, in order to help a transaction commit, the proposed construction allows a transaction to be executed and committed by a processor different the one it originated from. The idea to execute a transaction on a different processor has already been proposed in [26] and [27] where this idea is used to provide processes with a type of contention management refereed to as *transactional scheduling*. A very short description of these papers has been given in the introduction. Like contention managers, these schedulers can provide progress, but none of them ensure the progress of a process with a transaction that conflicts with some other transaction which has reached a point at which it must not be aborted. This means that the progress of a process still depends on the progress of another process.

Obstruction-Freedom, Lock-Freedom Many STM systems have been proposed that do not use locks. Some of them are obstruction-free (e.g., [80, 57]) or lock-free (in the sense there is no deadlock) [38]. Unfortunately, none of them provides the property that every issued transaction is committed. At most, the transactions that are helped in these papers are the ones in the process of committing. It can easily be seen that, with only this type of help, some transaction can be aborted indefinitely.

In an obstruction-free STM system a transaction that is executed alone must eventually commit. So consider some transaction that is always stalled (before its commit operation) and, while it is stalled, some conflicting transaction commits. It is easy to build an execution in which this stalled transaction never commits.

In a lock-free STM system, infinitely many transaction invocations must commit in an infinite execution. Again it is possible to build an execution in which a transaction is always stalled (before its commit operation) and is aborted by a concurrent transaction (transactions cannot wait for this stalled transaction because they do not know if it is making progress).

An interesting discussion on the possible/impossible liveness properties of a STM system is presented in [41] where is also described a general lock-free STM system. This system is based on a mechanism similar to the Compare&Swap used in this paper.

Irrevocable Transactions The aim of the concept of *irrevocable* (or *inevitable*) transaction is to provide the programmer with a special transaction type (or tag) related to its liveness or progress. Ensuring that a transaction does not abort is usually required for transactions that perform some operations that cannot be rolled back or aborted such as I/O. In order to solve this issue, certain STM systems provide irrevocable transactions which will never be aborted once they are typed irrevocable. This is done by preventing concurrent conflicting transactions from committing when an irrevocable transaction is being executed.

It is interesting to note that (a) an irrevocable transaction must be run exactly once and (b) only one irrevocable transaction can be executed at a time in the system (unless the shared memory accesses of the transaction are known ahead of time). Possible solutions are proposed and discussed in [56] and [60]. Irrevocable transactions suited to deadline-aware scheduling are presented in [52].

The STM system proposed in this paper does not support irrevocable transactions. Since an irrevocable transaction cannot be executed more than once, it is obvious that irrevocable transactions cannot benefit from the helping mechanism proposed in the paper. Our STM construction could nevertheless be extended to provide this functionality by having a process that wants to execute an irrevocable transaction issue a Compare&Swap on an irrevocability flag to the end of the list in order to prevent any other concurrent transaction from committing until it commits itself. Unfortunately, this would violate the liveness and progress properties previously guaranteed. That is why in our model, "irrevocable transactions" appear as non-transactional code.

Robust STMs Ensuring progress even when bad behavior (such as process crash) can occur has been investigated in several papers. As an example, [59] presents a robust STM system where a transaction that is not committed for a too long period eventually gets priority using locks. It is assumed that the system provides a crash detection mechanism that allows locks to be stolen once a crash is detected. This paper also presents a technique to deal with non-terminating transactions.

Universal construction for concurrent objects The notion of a universal construction for concurrent objects (or concurrent data structures) has been introduced by Herlihy [43]. Since then, several universal constructions have been proposed. (e.g., [24, 25, 33]). The concurrent objects suited to such constructions are the objects that are defined by a sequential specification on total operations (i.e., operations that, when executed alone, always return a result).

There an important and fundamental difference between an operation on a concurrent object (e.g., a shared queue) and a transaction. Albeit the operations on a queue can have many different implementations, their semantics is defined once for all. Differently, each transaction is a specific atomic procedure whose code can be seen as being any dynamically defined code.

A universal construction for wait-free *transaction friendly* concurrent objects is presented in [29]. The words “transaction friendly” means here that a process that has invoked an operation on an object can abort it during its execution. Hence, a ‘transaction friendly” concurrent object is a kind of abortable object. It is important to notice that this abortion notion is different from the notion of transaction abortion. In the first case, the abort of an operation is a programming level notion that the construction has to implement. Differently, in the second case, a transaction abort is due the implementation itself. More precisely, transaction abortion is then a system level mechanism used to prevent global inconsistency when the system allows concurrent transactions to be executed optimistically (differently, albeit very inefficient, using a single global lock for all transactions would allow any transaction to executed without being aborted). (A main issue solved in [29] lies in ensuring that, without violating wait-freedom, an operation *op* issued by a process p_i is not committed in the back of p_i by another process p_j which is helping it to execute that operation.)

3.7 Computation models

This section presents the programming model offered to the programmers and the underlying multiprocessor model on top of which the universal STM system is built.

3.7.1 The user programming model

The program written by the user is made up of n sequential processes denoted p_1, ..., p_n. Each process is a sequence of transactions in which two consecutive transactions can be separated by non-transactional code. Both transactions and non-transactional code can access concurrent objects.

Transactions A transaction is an atomic unit of computation (atomic procedure) that can access concurrent objects called t-objects. “Atomic” means that (from the programmer’s point of view) a transaction appears as being executed instantaneously at a single point of the time line (between its start event and its end event) and no two transactions are executed at the same point of the time line. It is assumed that, when executed alone, a transaction always terminates.

Non-transactional code Non-transactional code is made up of statements for which the user does not require them to appear as being executed as a single atomic computation unit. This code usually contains input/output statements (if any). Non-transactional code can also access concurrent objects. These objects are called *nt*-objects.

Concurrent objects Concurrent objects shared by processes (user level) are denoted with small capital letters. It is assumed that a concurrent object is either an *nt*-object or a *t*-object (not both). Moreover, each concurrent object is assumed to be linearizable.

The atomicity property associated with a transaction guarantees that all its accesses to *t*-objects appear as being executed atomically. As each concurrent object is linearizable (i.e., atomic), the atomicity power of a transaction is useless if the transaction only accesses a single *t*-object once. Hence encapsulating accesses to concurrent objects in a single transaction is "meaningful" only if that transaction accesses several objects or accesses the same object several times (as in a Read/Modify/Write operation).

As an example let us consider a concurrent queue (there are very efficient implementation of such an object, e.g., [53]). If the queue is always accessed independently of the other concurrent objects, its accesses can be part of non-transactional code and this queue instance is then an *nt*-object. Differently, if the queue is used with other objects (for example, when moving an item from a queue to another queue) the corresponding accesses have to be encapsulated in a transaction and the corresponding queue instances are then *t*-objects.

Semantics As already indicated the properties offered to the user are (1) linearizability (safety) and (2) the fact that each transaction invocation entails exactly one execution of that transaction (liveness).

3.7.2 The underlying system model

The underlying system is made up of m processors (simulators) denoted P_1, ..., P_m. We assume $n \geq m$. The processors communicate through shared memory that consists of single-writer/multi-reader (1WMR) atomic registers, compare&swap registers and fetch&increment registers.

Notation The objects shared by the processors are denoted with capital italic letters. The local variables of a processor are denoted with small italic letters.

Compare&swap register A compare&swap register X is an atomic object that provides processors with a single operation denoted X.Compare&Swap(). This operation is a conditional write that returns a boolean value. Its behavior can be described by the following statement:

```
operation $X$.Compare&Swap($old, new$):
    $atomic\{$ if $X = old$ then $X \leftarrow new$; return($true$) else return($false$) end if. $\}$
```

Fetch&increment register A fetch&increment register X is an atomic object that provides processors with a single operation, denoted X.Fetch&Increment(), that adds 1 to X and returns its new value.

3.8 A universal construction for STM systems

This section describes the proposed universal construction. It first introduces the control variables shared by the m processors and then describes the construction. As already indicated, its design is based on simple principles: (1) each processor is assigned a subset of processes for which it is in charge of their individual progress; (2) when a processor does not succeed in executing and committing a transaction issued by a process it owns, it requires help from the other processors; (3) the state of the t-objects accessed by transactions is represented by a list that is shared by the processors (similarly to [43]).

Without loss of generality, the proposed construction considers that the concurrent objects shared by transactions (t-objects) are atomic read/write objects. Extending to more sophisticated linearizable concurrent objects is possible. We limit our presentation to atomic read/write objects to keep it simpler.

In our universal construction, the STM controls entirely the transactions; this is different from what is usually assumed in STMs [41].

3.8.1 Control variables shared by the processors

This section presents the shared variables used by the processors to execute the multiprocess program. Each processor also has local variables which will be described when presenting the construction.

Pointer notation Some variables manipulated by processors are pointers. The following notation is associated with pointers. Let PT be a pointer variable. $\downarrow PT$ denotes the object pointed to by PT. let OB be an object. $\uparrow OB$ denotes a pointer to OB. Hence, $\uparrow (\downarrow PT) = PT$ and $\downarrow (\uparrow OB) = OB$.

Process ownership Each processor P_x is assigned a set of processes for which it has the responsibility of ensuring individual progress. A process p_i is assigned to a single processor. We assume here a static assignment. (It is possible to consider a dynamic process assignment. This would require an appropriate underlying scheduler. We do not consider such a possibility here in order to keep the presentation simple.)

The process assignment is defined by an array $OWNED\_BY[1..m]$ such that the entry $OWNED\_BY[x]$ contains the set of identities of the processes "owned" by processor P_x. As we will see below the owner P_x of process p_i can ask other processors to help it execute the last transaction issued by p_i.

Representing the state of the t-objects As previously indicated, at the processor (simulation) level, the state of the t-objects of the program is represented by a list of descriptors such that each descriptor is associated with a transaction that has been committed.

FIRST is a compare&swap register containing a pointer to the first descriptor of the list. Initially *FIRST* points to a list containing a single descriptor associated with a fictitious transaction that gives an initial value to each t-object. Let *DESCR* be the descriptor of a (committed) transaction T. It has the following four fields.

- *DESCR.next* and *DESCR.prev* are pointers to the next and previous items of the list.

- $DESCR.tid$ is the identity of T. It is a pair $\langle i, t\_sn \rangle$ where i is the identity of the process that issued the transaction and t_sn is its sequence number (among all transactions issued by p_i).
- $DESCR.ws$ is a set of pairs $\langle \text{X}, v \rangle$ stating that T has written v into the concurrent object X.
- $DESCR.local\_state$ is the local state of the process p_i just before the execution of the transaction or the non-transactional code that follows T in the code of p_i.

Helping mechanism: the array $LAST\_CMT[1..m, 1..n]$ This array is such that $LAST\_CMT[x, i]$ contains the sequence number of process p_i's last committed transaction as known by processor P_x. $LAST\_CMT[x, i]$ is written only by P_x. Is initial value is 0.

Helping mechanism: logical time $CLOCK$ is an atomic fetch&increment register initialized to 0. It is used by the helping mechanism to associate a logical date with a transaction that has to be helped. Dates define a total order on these transactions. They are used to ensure that any helped transaction is eventually committed.
Helping mechanism: the array $STATE[1..n]$ This array is such that $STATE[i]$ describes the current state of the execution (simulation) of process p_i. It has four fields.

- $STATE[i].tr\_sn$ is the sequence number of the next transaction to be issued by p_i.
- $STATE[i].local\_state$ contains the local state of p_i immediately before the execution of its next transaction (whose sequence number is currently kept in $STATE[i].tr\_sn$).
- $STATE[i].help\_date$ is an integer (date) initialized to $+\infty$. The processor P_x (owner of process p_i) sets $STATE[i].help\_date$ to the next value of $CLOCK$ when it requires help from the other processors in order for the last transaction issued by p_i to be eventually committed.
- $STATE[i].last\_ptr$ contains a pointer to a descriptor of the transaction list (its initial value is $FIRST$). $STATE[i].last\_ptr = pt$ means that, if the transaction identified by $\langle i, STATE[i].tr\_sn \rangle$ belongs to the list of committed transactions, it appears in the transaction list after the transaction pointed to by pt.

3.8.2 How the t-objects and nt-objects are represented

Let us remember that the t-objects and nt-objects are the objects accessed by the processes of the application program. The nt-objects are directly implemented in the memory shared by the processors and consequently their operations access directly that memory.

Differently, the values of the t-objects are kept in the ws field of the descriptors associated with committed transactions (these descriptors define the list pointed to by $FIRST$). More precisely, we have the following.

- A write of a value v into a t-object X by a transaction appears as the pair $\langle \text{X}, v \rangle$ contained in the field ws of the descriptor that is added to the list when the corresponding transaction is committed.

- A read of a t-object X by a transaction is implemented by scanning downwards (from a fixed local pointer variable *current* towards *FIRST*) the descriptor list until encountering the first pair $\langle X, v \rangle$, the value v being then returned by the read operation. It is easy to see that the values read by a transaction are always mutually consistent (if the values v and v' are returned by the reads of X and Y issued by the same transaction, then the first value read was not overwritten when the second one was read).

3.8.3 Behavior of a processor: initialization

Initially a processor P_x executes the non-transactional code (if any) of each process p_i it owns until p_i's first transaction and then initializes accordingly the atomic register $STATE[i]$. Next P_x invokes $\mathsf{select}(OWNED\_BY[x])$ that returns the identity of a process it owns, this value is then assigned to P_x's local variable *my_next_proc*. P_x also initializes local variables whose role will be explained later. This is described in Figure 3.1.

The function $\mathsf{select}(set)$ is *fair* in the following sense: if it is invoked infinitely often with $i \in set$, then i is returned infinitely often (this can be easily implemented). Moreover, $\mathsf{select}(\emptyset) = \perp$.

```
for each $i \in OWNED\_BY[x]$ do
    execute $p_i$ until the beginning of its first transaction;
    $STATE[i] \leftarrow \langle 1, p_i\text{'s current local state}, +\infty, FIRST \rangle$
end for;
$my\_next\_proc \leftarrow \mathsf{select}(OWNED\_BY[x])$;
$k1\_counter \leftarrow 0$; $my\_last\_cmt$ is a pointer initialized to $FIRST$.
```

Figure 3.1: Initialization for processor P_x ($1 \le x \le m$)

3.8.4 Behavior of a processor: main body

The behavior of a processor P_x is described in Figure 3.2. This consists of a while loop that terminates when all transactions issued by the processes owned by P_x have been successfully executed. This behavior can be decomposed into 4 parts.

Select the next transaction to execute (Lines 01-11) Processor P_x first reads (asynchronously) the current progress of each process and selects accordingly a process (lines 01-02). The procedure $\mathsf{select\_next\_process}()$ (whose details will be explained later) returns it the identity i of the process for which P_x has to execute the next transaction. This process p_i can be a process owned by P_x or a process whose owner P_y requires the other processors to help it execute its next transaction.

Next, P_x initializes local variables in order to execute p_i's next transaction in the appropriate correct context (lines 03-05). Before entering a speculative execution of the transaction, P_x first looks to see if it has not yet been committed (lines 07-11). To that end, P_x scans the list of committed transactions. Thanks to the pointer value kept in $STATE[i].last\_ptr$, it is useless to scan the list from the beginning: instead the scan may start from the transaction descriptor pointed to by $current = state[i].last\_ptr$. If P_x discovers that the transaction has been previously committed it sets the boolean $committed$ to $true$.

It is possible that, while the transaction is not committed, P_x loops forever in the list because the predicate $(\downarrow current).next = \perp$ is never true. This happens when new committed transactions (different from P_x's transactions) are repeatedly and infinitely added to the list. The procedure prevent_endless_looping() (line 07) is used to prevent such an infinite looping. Its details will be explained later.

Speculative execution of the selected transaction (Lines 12-17) The identity of the transaction selected by P_x is $\langle i, i\_tr\_sn\rangle$. If, from P_x's point of view, this transaction is not committed, P_x simulates locally its execution (lines 13-17). The set of concurrent t-objects read by p_i is saved in P_x's local set *lrs*, and the pairs $\langle$Y,$v\rangle$ such that the transaction issued Y.write(v) are saved in the local set *ws*. This is a transaction's speculative execution by P_x.

Try to commit the transaction (Lines 18-32) Once P_x has performed a speculative execution of p_i's last transaction, it tries to commit it by adding it to the descriptor list, but only if certain conditions are satisfied. To that end, P_x enters a loop (lines 19-24). There are two reasons not to try to commit the transaction.

- The first is when the transaction has already been committed. If this is the case, the transaction appears in the list of committed transactions (scanned by the pointer *current*, lines 21-22).
- The second is when the transaction is an update transaction and it has read a t-object that has then been overwritten (by a committed transaction). This is captured by the predicate at line 23.

Then, if (a) the transaction has not yet been committed (as far as P_x knows) and (b1) no t-object read has been overwritten or (b2) the transaction is read-only, then P_x tries to commit its speculative execution of this transaction (line 25). To do this it first creates a new descriptor *DESCR*, updates its fields with the data obtained from its speculative execution (line 27) and then tries to add it to the list. To perform the commit, P_x issues Compare&Swap$((\downarrow current).next, \perp, \uparrow DESCR)$. It is easy to see that this invocation succeeds if and only if *current* points to the last descriptor of the list of committed transactions (line 28).

Finally, if the transaction has been committed P_x updates $LAST\_CMT[x,i]$ (line 32).

Use the ownership notion to ensure the progress of each process (Lines 33-46) The last part of the description of P_x's behavior concerns the case where P_x is the owner of the process p_i that issued the current transaction selected by P_x (determined at line 03). This means that P_x is responsible for guaranteeing the individual progress of p_i. There are two cases.

- If *committed* is equal to *false*, P_x requires help from the other processors in order for p_i's transaction to be eventually committed. To that end, it assigns (if not yet done) the next date value to $STATE[i].help\_date$ (lines 35-38). Then, P_x proceeds to the next loop iteration. (Let us observe that, in that case, *my_next_proc* is not modified.)
- Given that P_x is responsible for p_i's progress, if *committed* is equal to *true* then P_x executes the non-transactional code (if any) that appears after the transaction (line 39). Next, if p_i has terminated (finished its execution), i is suppressed from $OWNED\_BY[x]$ (line 41). Otherwise, P_x updates $STATE[i]$ in order for it to contain the information required to execute the next transaction of p_i (line 42). Finally, before re-entering the main loop, P_x updates the pointer *my_last_cmt* (see below) and *my_next_proc* in order to ensure the progress of the next process it owns (line 44).

```
while ($my\_next\_proc \neq \bot$) do
      % — Selection phase ————————————————————————————
(01) $state[1..n] \leftarrow [STATE[1], \cdots, STATE[n]]$;
(02) $i \leftarrow$ select_next_process();
(03) $i\_local\_state \leftarrow state[i].local\_state$; $i\_tr\_sn \leftarrow state[i].tr\_sn$;
(04) $current \leftarrow state[i].last\_ptr$; $committed \leftarrow false$;
(05) $k2\_counter \leftarrow 0$; $after\_my\_last\_cmt \leftarrow false$;
(06) while $\big( ((\downarrow current).next \neq \bot) \wedge (\neg committed) \big)$ do
(07)        prevent_endless_looping($i$);
(08)        if $((\downarrow current).tid = \langle i, i\_tr\_sn \rangle)$
(09)           then $committed \leftarrow true$; $i\_local\_state \leftarrow (\downarrow current).local\_state$ end if;
(10)        $current \leftarrow (\downarrow current).next$
(11) end while;
(12) if ($\neg committed$) then
        % — Simulation phase ———————————————————————————-
(13)    execute the $i\_tr\_sn$-th transaction of $p_i$: the value of X.read() is obtained
(14)    by scanning downwards the transaction list (starting from $current$);
(15)    $p_i$'s local variables are read from (written into) $P_x$'s local memory (namely, $i\_local\_state$);
(16)    The set of shared objects read by the current transaction are saved in the set $lrs$;
(17)    The pairs $\langle$Y,$v\rangle$ such that the transaction issued Y.write($v$) are saved in the set $ws$;
        % — Try to commit phase ————————————————————————
(18)    $overwritten \leftarrow false$;
(19)    while $\big( (\downarrow current).next \neq \bot) \wedge (\neg committed) \big)$ do
(20)          prevent_endless_looping($i$);
(21)          $current \leftarrow (\downarrow current).next$;
(22)          same as lines 08 and 09;
(23)          if ($\exists$ X $\in lrs$ : $\langle$X$, - \rangle \in (\downarrow current).ws$) then $overwritten \leftarrow true$ end if
(24)    end while;
(25)    if ($\neg committed \wedge (\neg overwritten \vee ws = \emptyset)$)
(26)       then allocate a new transaction descriptor $DESCR$;
(27)            $DESCR \leftarrow \langle \bot, current, \langle i, i\_tr\_sn \rangle, ws, i\_local\_state \rangle$;
(28)            $committed \leftarrow$ Compare&Swap($(\downarrow current).next, \bot, \uparrow DESCR$);
(29)            if ($\neg committed$) then disallocate $DESCR$ end if
(30)    end if
(31) end if;
(32) if ($committed$) then $LAST\_CMT[x, i] \leftarrow i\_tr\_sn$ end if;
        % — End of transaction ——————————————————————————
(33) if ($i \in OWNED\_BY[x]$) then
(34)    if ($\neg committed$)
(35)       then if ($state[i].help\_date = +\infty$) then
(36)               $helpdate \leftarrow$ Fetch&Incr($CLOCK$);
(37)               $STATE[i] \leftarrow \langle state[i].tr\_sn, state[i].local\_state, helpdate, state[i].last\_ptr \rangle$
(38)            end if
(39)       else execute non-transactional code of $p_i$ (if any) in the local context $i\_local\_state$;
(40)            if (end of $p_i$'s code)
(41)               then $OWNED\_BY[x] \leftarrow OWNED\_BY[x] \setminus \{i\}$
(42)               else $STATE[i] \leftarrow \langle i\_tr\_sn + 1, i\_local\_state, +\infty, current \rangle$
(43)            end if;
(44)            $my\_last\_cmt \leftarrow \uparrow DESCR$; $my\_next\_proc \leftarrow$ select($OWNED\_BY[x]$)
(45)    end if
(46) end if
end while.
```

Figure 3.2: Algorithm for processor P_x ($1 \leq x \leq m$)

3.8.5 Behavior of a processor: starvation prevention

Any transaction issued by a process has to be eventually executed by a processor and committed. To that end, the helping mechanism introduced previously has to be enriched so that no processor either (a) permanently helps only processes owned by other processors or (b) loops forever in an internal while loop (lines 06-11 or 19-24). The first issue is solved by procedure select_next_process() while the second issue is solved by the procedure prevent_endless_looping().

Each of these procedures uses an integer value (resp., $K1$ and $K2$) as a threshold on the length of execution periods. These periods are measured with counters (resp., $k1\_counter$ and $k2\_counter$). When one of these periods attains its threshold, the corresponding processor requires help for its pending transaction. The values $K1$ and $K2$ can be arbitrary. They could be tuned or even defined dynamically in order to make the construction more efficient.

The procedure select_next_process() This operation is described in Figure 3.3. It is invoked at line 02 of the main loop and returns a process identity. Its aim is to allow the invoking processor P_x to eventually make progress for each of the processes it owns.

The problem that can occur is that a processor P_x can permanently help other processors execute and commit transactions of the processes they own, while none of the processes owned by P_x is making progress. To prevent this bad scenario from occurring, a processor P_x that does not succeed in having its current transaction executed and committed for a "too long" period, requires help from the other processors.

```
procedure select_next_process() returns (process id) =
(101) let $set = \{ i \mid (state[i].help\_date \neq +\infty) \wedge$
                  $( (\forall y: LAST\_CMT[y,i] < state[i].tr\_sn) \vee (i \in OWNED\_BY[x]) )\}$;
(102) if $(set = \emptyset)$
(103)    then $i \leftarrow my\_next\_proc$; $k1\_counter \leftarrow 0$
(104)    else $i \leftarrow \min(set)$ computed with respect to transaction help dates;
(105)       if $(i \in OWNED\_BY[x])$
(106)          then $k1\_counter \leftarrow 0$
(107)          else $k1\_counter \leftarrow k1\_counter + 1$;
(108)             if $(k1\_counter \geq K1)$ then
(109)                let $j = my\_next\_proc$;
(110)                if $(state[j].help\_date = +\infty)$ then
(111)                   $helpdate \leftarrow$ Fetch&Incr$(CLOCK)$;
(112)                   $STATE[j] \leftarrow \langle state[j].tr\_sn, state[j].local\_state, helpdate, state[j].last\_ptr\rangle$
(113)                end if;
(114)                $k1\_counter \leftarrow 0$
(115)             end if
(116)       end if
(117) end if;
(118) return$(i)$.
```

Figure 3.3: The procedure select_next_process()

This is realized as follows. P_x first computes the set set of processes p_i for which help has been required (those are the processes whose help date is $\neq +\infty$) and, (as witnessed by the array $LAST\_CMT$) either no processor has yet publicized the fact that their last transactions have been committed or p_i is owned by P_x (line 101). If set is empty (no help is required), select_next_process() returns the identity of the next process owned by P_x (line 103). If $set \neq \emptyset$, there are processes to help and P_x selects the identity i of the process with the oldest help date (line 104). But before returning the identity i (line 118), P_x checks if it has been waiting for a too long period before having its next transaction executed. There are then two cases.

- If $i \in OWNED\_BY[x]$, P_x has already required help for the process p_i for which it strives to make progress. It then resets the counter $k1\_counter$ to 0 and returns the identity i (line 106).
- If $i \notin OWNED\_BY[x]$, P_x first increases $k1\_counter$ (line 107) and checks if it attains its threshold $K1$. If this is the case, the logical period of time is too long (line 109) and consequently (if not yet done) P_x requires help for the last transaction of the process p_j (such that $my\_next\_proc = j$). As we have seen, "require help" is done by assigning the next clock value to $STATE[j].help\_date$ (lines 109-113). In that case, P_x also resets $k1\_counter$ to 0 (line 114).

```
procedure prevent_endless_looping($i$);
(201) if ($i \in OWNED\_BY[x]$) then
(202)    if ($current$ has bypassed $my\_last\_cmt$) then $k2\_counter \leftarrow k2\_counter + 1$ end if;
(203)    if $\big((k2\_counter > K2) \wedge (state[i].help\_date = +\infty)\big)$
(204)       then $helpdate \leftarrow$ Fetch&Incr($CLOCK$);
(205)            $STATE[i] \leftarrow \langle state[i].tr\_sn, state[i].local\_state, helpdate, state[i].last\_ptr\rangle$
(206)    end if
(207) end if.
```

Figure 3.4: Procedure prevent_endless_looping()

The procedure prevent_endless_looping() As indicated, the aim of this procedure, described in Figure 3.4, is to prevent a processor P_x from endless looping in an internal while loop (lines 05-09 or 17-21).

The time period considered starts at the last committed transaction issued by a process owned by P_x. It is measured by the number of transactions committed since then. The beginning of this time period is determined by P_x's local pointer *my_last_cmt* (which is initialized to *FIRST* and updated at line 44 of the main loop after the last transaction of a process owned by P_x has been committed.)

The relevant time period is measured by processor P_x with its local variable $k2\_counter$. If the process p_i currently selected by select_next_process() is owned by P_x (line 251), then P_x will require help for p_i once this period attains $K2$ (lines 253-256). In that way, the transaction issued by that process will be executed and committed by other processors and (if not yet done) this will allow P_x to exit the while loop because its local boolean variable *committed* will then become true (line 09 of the main loop).

3.9 Proof of the STM construction

Let *PROG* be a transaction-based n-process concurrent program. The proof of the universal construction consists in showing that a simulation of *PROG* by m processors that execute the algorithms described in Figures 3.1-3.4 generates an execution of *PROG*.

Lemma 12 *Let T be the transaction invocation with the smallest help date (among all the transaction invocations not yet committed for which help has been required). Let p_i be the process that issued T and P_y a processor. If T is never committed, there is a time after which P_y issues an infinite number invocations of* select_next_process() *and they all return i.*

Proof Let us assume by contradiction that there is a time after which either P_y is blocked within an internal while loop (Figure 3.2) or its invocations of select_next_process() never return i. It follows from line 104 of select_next_process() that the process identity of the transaction from *set* with the smallest help date is returned. This means that for i to never be returned, there must always be some transaction(s) in *set* with a smaller help date than T. By definition we know that T is the uncommitted transaction with the smallest help date, so any transaction(s) in *set* with a smaller help date must be already committed. Let us call this subset of committed transactions T_{set}. Since *set* is finite, T_{set} also is finite. Moreover, T_{set} cannot grow because any transaction T' added to the array $STATE[1..n]$ has a larger help date than T (such a transaction T' has asked for help after T and due to the Fetcch&Increment() operation the help dates are monotonically increasing). So to complete the contradiction we need to show that (a) P_y is never blocked forever in an internal while loop (Figure 3.2) and (b) eventually $T_{set} = \emptyset$.

If T_{set} is not empty, select_next_process() returns the process identity j for some committed transaction $T' \in T_{set}$. On line 09, the processor P_y will see T' in the list and perform $committed \leftarrow true$. Hence, P_y cannot block forever in an internal while loop. Then, on line 32, P_y updates $LAST\_CMT[y, j]$. Let us observe that, during the next iteration of select_next_process() by P_x, T' is not be added to *set* (line 101) and, consequently, there is then one less transaction in T_{set}. And this continues until T_{set} is empty. After this occurs, each time processor P_y invokes select_next_process(), it obtains the process identity i, which invalidates the contradiction assumption and proves the lemma. $\Box_{Lemma\ 12}$

Lemma 13 *Any invocation of a transaction T that requests help (hence it has $helpdate \neq \infty$) is eventually committed.*

Proof Let us first observe that all transactions that require help have bounded and different help dates (lines 36-37, 111-112 or 255-256). Moreover, once defined, the helping date for a transaction is not modified.

Among all the transactions that have not been committed and require help, let T be the transaction with the smallest help date. Assume that T has been issued by process p_i owned by processor P_x (hence, P_x has required help for T). Let us assume that T is never committed. The proof is by contradiction.

As T has the smallest help date, it follows from Lemma 12 that there is a time after which all the processors that call select_next_process() obtains the process identity i. Let $\mathscr{P}$ be this non-empty set of processors. (The other processors are looping in a while loop or are slow.) Consequently, given that all transactions that are not slow are trying to commit T (by performing a compare&swap() to add it to the list), that the list is not modified anywhere else, and that we assume that T never commits, there is a finite time after which the descriptor list does no longer increase. Hence, as the predicate $(\downarrow current).next = \perp$ becomes eventually true, we conclude that at least one processor $P_y \in \mathscr{P}$ cannot be blocked forever in a while loop. Because the list is no longer changing, the predicate of line 25 then becomes satisfied at P_y. It follows that, when the processors of $\mathscr{P}$ execute line 28, eventually one of them successfully executes the compare&swap that commits the transaction T which contradicts the initial assumption.

As the helping dates are monotonically increasing, it follows that any transaction T that requires help is eventually committed. $\Box_{Lemma\ 13}$

Lemma 14 *No processor P_x loops forever in an internal while loop (lines 06-11 or 19-24).*

Proof The proof is by contradiction. Let P_y be a processor that loops forever in an internal while loop. Let i be the process identity it has obtained from its last call to select_next_process() (line 02) and P_x be the processor owner of p_i.

Let us first show that processor P_x cannot loop forever in an internal while loop. Let us assume the contrary. Because processor P_x loops forever we never have $((\downarrow current).next = \bot) \vee committed$, but each time it executes the loop body, P_x invokes prevent_endless_looping(i) (at line 07 or 20). The code of this procedure is described in Figure 3.4. As $i \in OWNED\_BY[i]$ and P_x invokes infinitely often prevent_endless_looping(i), it follows from lines 251-253 and the current value of *my_last_cmt* (that points to the last committed transaction issued by a process owned by P_x, see line 44) that P_x's local variable $k2\_counter$ is increased infinitely often. Hence, eventually this number of invocations attains $K2$. When this occurs, if not yet done, P_x requires help for the transaction issued by p_i (lines 253-256). It then follows from Lemma 13, that p_i's transaction T is eventually committed. As the pointer *current* of P_x never skips a descriptor of the list and the list contains all and only committed transactions, we eventually have $(\downarrow current).tid = \langle i, i\_tr\_sn \rangle$ (where i_tr_sn is T's sequence number among the transactions issued by p_i). When this occurs, P_x's local variable *committed* is set to *true* and P_x stops looping in an internal while loop.

Let us now consider the case of a processor $P_y \neq P_x$. Let us first notice that the only way for P_y to execute T is when T has requested help (line 101 of operation select_next_process()). The proof follows from the fact that, due to Lemma 13, T is eventually committed. As previously (but now *current* is P_y's local variable), the predicate $(\downarrow current).tid = \langle i, i\_tr\_sn \rangle$ eventually becomes true and processor P_y sets *committed* to *true*. P_y then stops looping inside an internal while loop (line 08 or 22) which concludes the proof of the lemma. $\Box_{Lemma\ 14}$

Lemma 15 *Any invocation of a transaction T by a process is eventually committed.*

Proof Considering a processor P_x, let $i \in OWNED\_BY[x]$ be the current value of its local control variable *my_next_proc*. Let T be the current transaction issued by p_i. We first show that T is eventually committed.

Let us first observe that, as p_i has issued T, P_x has executed line 42 where it has updated $STATE[i]$ that now refers to that transaction. If P_x requires help for T, the result follows from Lemma 13. Hence, to show that T is eventually committed, we show that, if P_x does not succeed in committing T without help, it necessarily requires help for it. This follows from the code of the procedure select_next_proc(). There are two cases.

- select_next_process() returns i. In that case, as P_x does not loop forever in a while loop (Lemma 14), it eventually executes lines 33-38 and consequently either commits T or requires help for T at line 37.
- select_next_process() never returns i. In that case, as P_x never loops forever in a while loop (Lemma 14), it follows that it repeatedly invokes select_next_process() and, as these invocations do not return i, the counter $k1\_counter$ repeatedly increases and eventually

attains the value $K1$. When this occurs P_x requires help for T (lines 107-115) and, due to Lemma 13, T is eventually committed.

Let us now observe that that, after T has been committed (by some processor), P_x executes lines 39-44 where it proceeds to the simulation of its next process (as defined by $\mathsf{select}(OWNED\_BY[x])$). It then follows from the previous reasoning that the next transaction of the process that is selected (whose identity is kept in *my_next_proc*) is eventually committed.

Finally, as the function $\mathsf{select}()$ is fair, it follows that no process is missed forever and, consequently, any transaction invocation issued by a process is eventually committed. $\Box_{Lemma\ 15}$

Lemma 16 *Any invocation of a transaction T by a process is committed at most once.*

Proof Let T be a transaction committed by a processor P_y (i.e., the corresponding $\mathsf{Compare\&Swap}()$ at line 28 is successful). T is identified $\langle i, STATE[i].ts\_sn\rangle$. As P_y commits T, we conclude that P_y has previously executed lines 06-28.

- We conclude from the last update of $STATE[i].last\_ptr = pt$ by P_y (line 42) and the fact that P_y's *current* local variable is initialized to $STATE[i].last\_ptr$, that T is not in the descriptor list before the transaction pointed to by pt.
- Let us consider the other part of the list. As T is committed by P_y, its pointer *current* progresses from $STATE[i].last\_ptr = pt$ until its last value that is such that $(\downarrow current).next = \perp$. It then follows from lines 08 and 22 that P_y has never encountered a transaction identified $\langle i, STATE[i].ts\_sn\rangle$ (i.e., T) while traversing the descriptor list.

It follows from the two previous observations that, when it is committed (added to the list), transaction T was not already in the list, which concludes the proof of the lemma. $\Box_{Lemma\ 16}$

Lemma 17 *Each invocation of a transaction T by a process is committed exactly once.*

Proof The proof follows directly from Lemma 15 and Lemma 16. $\Box_{Lemma\ 17}$

Lemma 18 *Each invocation of non-transactional code issued by a process is executed exactly once.*

Proof This lemma follows directly from lines 39-44: once the non-transactional code separating two transaction invocations has been executed, the processor P_x that owns the corresponding process p_i makes it progress to the beginning of its next transaction (if any). $\Box_{Lemma\ 18}$

Lemma 19 *The simulation is starvation-free (no process is blocked forever by the processors).*

Proof This follows directly from Lemma 14, Lemma 17, Lemma 18 and the definition of the function $\mathsf{select}()$. $\Box_{Lemma\ 19}$

Lemma 20 *The transaction invocations issued by the processes are linearizable.*

Proof To prove the lemma we have (a) to associate a linearization point with each transaction invocation and (b) show that the corresponding sequence of linearization points is consistent, i.e., the values read from t-objects by a transaction invocation T are uptodate (there have not been overwritten in that sequence). As far as item (a) is concerned, the linearization point of a transaction invocation is defined as follows[4].

- Update transactions (these are the transactions that write at least one t-object). The linearization point of the invocation of an update transaction is the time instant of the (successful) compare&swap statement that entails its commit.

- Read-only transactions. Let W be the set of update transactions that have written a value that has been read by the considered read-only transaction. Let τ_1 be the time just after the maximum linearization point of the invocations of the transactions in W and τ_2 be the time at which the first execution of the considered transaction has started. The linearization point of the transaction is then $\max(\tau_1, \tau_2)$.

To prove item (b) let us consider the order in which the transaction invocations are added to the descriptor list (pointed to by *FIRST*). As we are about to see, this list and the linearization order are not necessarily the same for read-only transaction invocations. Let us observe that, due to the atomicity of the compare&swap statement, a single transaction invocation at a time is added to the list.

Initially, the list contains a single fictitious transaction that gives an initial value to every t-object. Let us assume that the linearization order of all the transaction invocations that have been committed so far (hence they define the descriptor list) is consistent (let us observe that this is initially true). Let us consider the next transaction T that is committed (i.e., added to the list). As previously, we consider two cases. Let p_i be the process that issued T, P_x the processor that owns p_i and P_y the processor that commits T.

- The transaction is an update transaction (hence, $ws \neq \emptyset$). In that case, P_y has found $(\downarrow current).next = \perp$ (because the compare&sap succeeds) and at line 25, just before committing, the predicate $\neg committed \wedge \neg overwritten$ is satisfied.

  As *overwritten* is false, it follows that none of the values read by T has been overwritten. Hence, the reads and writes on t-objects issued by T can appear as having been executed atomically at the time of the compare&swap. Moreover, the values of the t-objects modified by T are saved in the descriptor attached to the list by the compare&swap and the global state of the t-objects is consistent (i.e., if not overwritten before, any future read of any of these t-objects obtains the value written by T).

  Let us now consider the local state of p_i (the process that issued T). There are two cases.

  – $P_x = P_y$ (the transaction is committed by the owner of p_i). In that case, the local state of p_i after the execution of T is kept in P_x's local variable *i_local_state* (line 15). After processor P_x has executed the non-transactional code that follows the invocation of T (if any, line 39), it updates $STATE[i].local\_state$ with the current value of *i_local_state* (if p_i had not yet terminated, line 42).

[4]The fact that a transaction invocation is *read-only* or *update* cannot always be statically determined. It can depend on the code of transaction (this occurs for example when a transaction behavior depends on a predicate on values read from t-objects). In our case, a read-only transaction is a transaction with an empty write set (which cannot be always statically determined by a compiler).

- $P_x \neq P_y$ (the processor that commits T and the owner of p_i are different processors). In that case, P_y has saved the new local state of p_i in $DESCR.local\_state$ (line 27) just before appending $DESCR$ at the end of the descriptor list.

  Next, thanks to the the predicate $i \in OWNED\_BY[x]$ in the definition of set at line 101, there is an invocation of select_next_process() by P_x that returns i. When this occurs, P_x discovers at line 09 or 22 that the transaction T has been committed by another processor. It then retrieves the local state of p_i (after execution of T) in $(\downarrow current).local\_state$, saves it in i_local_state and (as in the previous item) eventually writes it in $STATE[i].local\_state$ (line 42).

It follows that, in both cases, the value saved in $STATE[i].local\_state$ is the local state of p_i after the execution of T and the non-transactional code that follows T (if any).

- The transaction is a read-only transaction (hence, $ws = \emptyset$). In that case, T has not modified the state of the t-objects. Hence, we only have to prove that the new local state of p_i is appropriately updated and saved in $STATE[i].local\_state$.

  The proof is the same as for the case of an update transaction. The only difference lies in the fact that now it is possible to have $overwritten \wedge ws = 0$. If $overwritten$ is true, T can no longer be linearized at the commit point. That is why it is linearization point has been defined just after the maximum linearization point of the transactions it reads from (or the start of T if it happens later), which makes it linearizable.

$\Box_{Lemma\ 20}$

Lemma 21 *The simulation of a transaction-based n-process program by m processors (executing the algorithms described in Figures 3.1-3.4) is linearizable.*

Proof Let us first observe that, due to Lemma 20, The transaction invocations issued by the processes are linearizable, from which we conclude that the set of t-objects (considered as a single concurrent object TO) is linearizable. Moreover, by definition, every nt-object is linearizable.

As (a)linearizability is a *local* consistency property [135][5] and (b) TO is linearizable and every nt-object is linearizable, it follows that the execution of the multiprocess program is linearizable. $\Box_{Lemma\ 21}$

Theorem 4 *Let PROG be a transaction-based n-process program. Any simulation of PROG by m processors executing the algorithms described in Figures 3.1-3.4 is an execution of PROG.*

Proof A formal statement of this proof requires an heavy formalism. Hence we only give a sketch of it. Basically, the proof follows from Lemma 19 and Lemma 21. The execution of *PROG* is obtained by projecting the execution of each processor on the simulation of the transactions it commits and the execution of the non-transactional code of each process it owns.
$\Box_{Theorem\ 4}$

[5]A property P is local if the set of concurrent objects (considered as a single object) satisfies P whenever each object taken alone satisfies P. It is proved in [135] that linearizability is a local property.

3.10 The number of tries is bounded

This section presents a bound for the maximum number of times a transaction can be unsuccessfully executed by a processor before being committed, namely, $O(m^2)$. A workload that has this bound is then given.

Lemma 22 *At any time and for any processor P_x, there is at most one atomic register $STATE[i]$ with $i \in OWNED\_BY[x]$ such that the corresponding transaction (the identity of which is $\langle i, STATE[i].tr\_sn \rangle$) is not committed and $STATE[i].help\_date \neq +\infty$.*

Proof Let us first notice that the help date of a transaction invoked by a process p_i can be set to a finite value only by the processor P_x that owns p_i. There are two places where P_x can request help.

- This first location is in the prevent_endless_looping() procedure. In that case, the transaction for which help is required is the last transaction invoked by process $p_{my\_next\_proc}$.
- The second location is on line 37 after the transaction invocation T aborts. It follows from line 103 of the operation select_next_process() that this invocation is also from the last transaction invoked by process $p_{my\_next\_proc}$.

So we only need to show that *my_next_proc* only changes when a transaction is committed, which follows directly from the predicates at lines 34 and 35 and the statements of line 44. $\Box_{Lemma\ 22}$

Theorem 5 *A transaction T invoked by a process p_i owned by processor P_x is tried unsuccessfully at most $O(m^2)$ times before being committed.*

Proof Let us first observe that a transaction T (invoked by a process p_i) is executed once before its help date is set to a finite value (if it is not committed after that execution). This is because only the owner P_x of p_i can select T (line 103) when its help date is $+\infty$. Then, after it has executed T unsuccessfully once, P_x requests help for T by setting its help date to a finite value (line 37).

Let us now compute how many times T can be executed unsuccessfully (i.e., without being committed) after its help date has been set to a finite value. As there are m processors and all are equal (as far as helping is concerned), some processor must execute T more than $O(m)$ times in order for T to be executed more than $O(m^2)$ times. We show that this is impossible. More precisely, assuming a processor P executes T, there are 3 cases that can cause this execution to be unsuccessful and as shown below each case can cause at most $O(m)$ aborts of T at P.

- Case 1. The first case is that some other transaction $T1$ that does not request help (its help date is $+\infty$) is committed by some other processor $P2$ causing P's execution of T to abort. Now by lines 102 and 103 after $P2$ commits $T1$, $P2$ will only be executing uncommitted transactions from the *STATE* array with finite help dates at least until T is committed, so any subsequent abort of T caused by $P2$ cannot be caused by $P2$ committing a transaction with $+\infty$ help date. So the maximum number of times this type of abort can happen from P is $O(1)$.

- Case 2. The second case is when some other uncommitted transaction $T1$ in the *STATE* array with a finite help date is committed by some other processor $P2$ causing T to abort. First by lemma 22 we know that there is a maximum of $m-1$ transactions that are not T that can be requesting help at this time and in order for them to commit before T they must have a help date smaller than T's. Also by lemma 16 we know that a transaction is committed exactly once so this conflict between $T1$ and T cannot occur again at $P2$. Now after committing $T1$, the next transaction (that asks for help) of a process that is owned by the same processor that owned $T1$ will have a larger help date than T so now there are only $m-1$ transactions that need help that could conflict with T. Repeating this we have at most $O(m)$ conflicts of this type for P.

- Case 3. The third case is that P's execution of T is aborted because some other process has already committed T. Then on line 08 P will see that T has been committed and not execute it again, so we have at most $O(1)$ conflicts of this type.

$\Box_{Theorem\ 5}$

The bound is tight The execution that is described below shows that a transaction T can be tried $O(m^2)$ times before being committed.

Let T be a transaction owned by processor $P(1)$ such that $P(1)$ executes T unsuccessfully once and requires help by setting its help date to a finite value. Now, let us assume that each of the $m-1$ other processors is executing a transaction it owns, all these transactions conflict with T and there are no other uncommitted transactions with their help date set to a finite value.

Now $P(1)$ starts executing T again, but meanwhile processor $P(2)$ commits its own transaction which causes T to abort. Next $P(1)$ and $P(2)$ each try to execute T, but meanwhile processor $P(3)$ commits its own transaction causing $P(1)$ and $P(2)$ to abort T. Next $P(1)$, $P(2)$, and $P(3)$ each try execute T, but meanwhile processor $P(4)$ commits its owns transaction causing $P(1)$, $P(2)$, and $P(3)$ to abort T. Etc. until processor $P(m-1)$ aborts all the execution of T by other processors, resulting in all m processor executing T. The transaction T is then necessarily committed by one of these final executions. So we have $1+1+2+3+\ldots+(m-1)+m$ trials of T which is $O(m^2)$.

3.11 A short discussion to conclude

The aim of the universal construction that has been presented was to demonstrate and investigate this type of construction for transaction-based multiprocess programs. (Efficiency issues would deserve a separate investigation.) To conclude, we list here a few additional noteworthy properties of the proposed construction.

- The construction is for the family of transaction-based concurrent programs that are time-free (i.e., the semantics of which does not depend on real-time constraints).

- The construction is lock-free and works whatever the concurrency pattern (i.e., it does not require concurrency-related assumption such as obstruction-freedom). It works for both finite and infinite computations and does not require specific scheduling assumptions. Moreover, it is independent of the fact that processes are transaction-free (they then share only *nt*-objects), do not have non-transactional code (they then share only *t*-objects accessed by transactions) or have both transactions and non-transactional code.

- The helping mechanism can be improved by allowing a processor to require help for a transaction only when some condition is satisfied. These conditions could be general or application-dependent. They could be static or dynamic and be defined in relation with an underlying scheduler or a contention manager. The construction can also be adapted to benefit from an underlying scheduling allowing the owner of a process to be dynamically defined.

  It could also be adapted to take into account *irrevocable* transactions [56, 60]. Irrevocability is an implementation property which can be demanded by the user for some of its transactions. It states that the corresponding transaction cannot be aborted (this can be useful when one wants to include inputs/outputs inside a transaction; notice that, in our model, inputs/outputs appear in non-transactional code).

- We have considered a failure-free system. It is easy to see that, in a crash-prone system, the crash of a processor entails only the crash of the processes it owns. The processes owned by the processors that do not crash are not prevented from executing.

In addition to the previous properties, the proposed construction helps better understand the atomicity feature offered by STM systems to users in order to cope with concurrency issues. Interestingly this construction has some "similarities" with general constructions proposed to cope with the net effect of asynchrony, concurrency and failures, such as the BG simulation [28] (where there are simulators that execute processes) and Herlihy's universal construction to build wait-free objects [43] (where an underlying list of consensus objects used to represent the state of the constructed object lies at the core of the construction). The study of these similarities would deserve a deeper investigation.

Chapter 4

Ensuring Strong Isolation in STM Systems

4.1 Isolation in STM Systems

4.2 A STM Protocol with Non-Blocking Non-Transactional Operations

4.2.1 Overview of TL2

4.2.2 The Protocol

4.2.3 ***Proof of Linearizability***

4.3 ***An Implementation***

4.4 Introduction

STM Systems. Transactional Memory (TM) [151, 160] has emerged as an attempt to allow concurrent programming based on sequential reasoning: By using TM, a user should be able to write a correct concurrent application, provided she can create a correct sequential program. The underlying TM system takes care of the correct implementation of concurrency. However, while most existing TM algorithms consider applications where shared memory will be accessed solely by code enclosed in a transaction, it still seems imperative to examine the possibility that memory is accessed both inside and outside of transactions.

Strong vs Weak Isolation. TM has to guarantee that transactions will be isolated from each other, but when it comes to transactions and non-transactional operations, there are two paths a TM system can follow: it may either act oblivious to the concurrency between transactions and non-transactional operations, or it may take this concurrency into account and attempt to provide isolation guarantees even between transactional and non-transactional operations. The first case is referred to as *weak isolation* while the second case is referred to as *strong isolation*. (This distinction of guarantees was originally made in [154], where reference was made to "weak atomicity" versus "strong atomicity".)

While weak isolation violates the isolation principle of the transaction abstraction, it could nevertheless be anticipated and used appropriately by the programmer, still resulting in correctly functioning applications. This would require the programmer to be conscious of eventual race conditions between transactional and non-transactional code that can change depending on the STM system used.

Desirable Properties. In order to keep consistent with the spirit of TM principles, however, a system should prevent unexpected results from occurring in presence of race conditions. Furthermore, concurrency control should ideally be implicit and never be delegated to the programmer [145, 155]. These are the reasons for which strong isolation is desirable. Under strong isolation, the aforementioned scenarios, where non-transactional operations violate transaction isolation, would not be allowed to happen. An intuitive approach to achieving strong isolation is to treat each non-transactional operation that accesses shared data as a "mini-transaction", i.e., one that contains a single operation. In that case, transactions will have to be consistent (see Sect. 4.5) not only with respect to each other, but also with respect to the non-transactional operations. However, while the concept of the memory transaction includes the possibility of abort, the concept of the non-transactional operation does not. This means that a programmer expects that a transaction might fail, either by blocking or by aborting. Non-transactional accesses to shared data, though, will usually be read or write operations, which the programmer expects to be atomic. While executing, a read or write operation is not expected to be de-scheduled, blocked or aborted.

Content of the Paper. This paper presents a TM algorithm which takes the previous issues into account. It is built on top of TM algorithm TL2 [148], a word-based TM algorithm that uses locks. More precisely, TL2 is modified to provide strong isolation with non-transactional read and write operations. However, the algorithm is designed without the use of locks for non-transactional code, in order to guarantee that their execution will always terminate. To achieve this, two additional functions are specified, which substitute conventional read or write operations that have to be performed outside of a transaction. Possible violations of correctness under strong isolation are reviewed in Sect. 4.5. The TL2 algorithm is described in Sect. 4.6. Section 4.7 describes the proposed algorithm that implements strong isolation for TL2, while Sect. 4.8 concludes the paper by summarizing the work and examining possible applications.

4.5 Correctness and Strong Isolation

Consistency Issues. Commonly, consistency conditions for TM build on the concept of *serializability* [157], a condition first established for the study of database transactions.

A concurrent execution of transactions is serializable, if there exists a serialization, i.e., a legal sequential execution equivalent to it. Serializability refers only to committed transactions, however, and fails to take into account the possible program exceptions that a TM transaction may cause - even if it aborts - when it observes an inconsistent state of memory.

Opacity [149], a stricter consistency condition for TM, requires that both committed as well as aborted transactions observe a consistent state of shared memory. This implies that in order for a concurrent execution of memory transactions to be opaque, there must exist an equivalent, legal sequential execution that includes both committed transactions and aborted transactions, albeit reduced to their read prefix. Other consistency conditions have also been proposed, such as *virtual world consistency* [152]. It is weaker than opacity while keeping its spirit (i.e., it depends on both committed transactions and aborted transactions).

Transaction vs Non-transactional Code. In a concurrent environment, shared memory may occasionally be accessed by both transactions as well as non-transactional operations. Traditionally, however, transactions are designed to synchronize only with other transactions without considering the possibility of non-transactional code; a program that accesses the same shared memory both transactionally and non-transactionally would be considered incorrect. A TM system that implements opacity minimally guarantees consistency between transactional accesses, however, consistency violations may still be possible in the presence of concurrent non-transactional code. Given this, it can still be acceptable to have concurrent environments that may be prone to some types of violations, as is the case with systems that provide weak isolation [154, 161]. Under weak isolation, transactional and non-transactional operations can be concurrent, but the programmer has to be aware of how to handle these. Interestingly, this possibility of *co-existence of two different paradigms* between strong and weak isolation reveals two different interpretations of transactional memory: On one hand considering TM as an implementation of shared memory, and, on the other hand, considering TM as an additional way of achieving synchronization, to be used alongside with locks, fences, and other traditional methods.

Under weak isolation, transactions are considered to happen atomically only with respect to other transactions. It is possible for non-transactional operations to see intermediate results of transactions that are still live. Conversely, a transaction may see the results of non-transactional operations that happened during the transaction's execution. If this behavior is not considered acceptable for an application, then the responsibility to prevent it is delegated to the programmer of concurrent applications for this system. However, in order to spare the programmer this responsibility, both the transactional memory algorithm as well as the non-transactional read and write operations must be implemented in a way that takes their co-existence into account. Such an implementation that provides synchronization between transactional and non-transactional code is said to provide strong isolation.

Providing Strong Isolation. There are different definitions in literature for strong isolation [154, 153, 150]. In this paper we consider strong isolation to be the following: (a) non-transactional operations are considered as "mini" transactions which never abort and contain only a single read or write operation, and (b) the consistency condition for transactions is opacity.

This definition implies that the properties that are referred to as *containment* and *non-interference* [154] are satisfied. Containment is illustrated in the left part of Fig. 4.1. There, under strong isolation, we have to assume that transaction T_1 happens atomically, i.e.,"all or nothing", also with respect to non-transactional operations. Then, while T_1 is alive, no non-transactional read, such as R_x, should be able to obtain the value written to x by T_1. Non-interference is illustrated in the right part of Fig. 4.1. Under strong isolation, non-transactional code should not interfere with operations that happen inside a transaction. Therefore, transaction T_1 should not be able to observe the effects of operations W_x and W_y, given that they happen concurrently with it, while no opacity-preserving serialization of T_1, W_x and W_y can be found. Non-interference violations can be caused, for example, by non-transactional operations that are such as to cause the ABA problem for a transaction that has read a shared variable x. An additional feature of strong isolation, implemented in this paper, is that non-transactional read and write operations never block or abort. For this reason, it is termed *terminating strong isolation*.

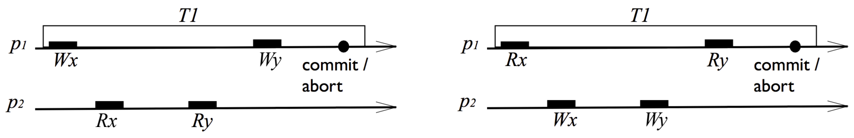


Figure 4.1: Left: *Containment* (operation R_x should not return the value written to x inside the transaction). Right: *Non-Interference* (wile it is still executing, transaction T_1 should not have access to the values that were written to x and y by process p_2).

Privatization/Publication. A discussion of the co-existence of transactional and non-transactional code would not be complete without mentioning the *privatization problem*. An area of shared memory is privatized, when a process that modified it makes it inaccessible to other concurrent processes[1] with the purpose being that the process can then access the memory without using synchronization operations [163]. A typical example of privatization would be the manipulation of a shared linked list. The removal of a node by a transaction T_i, for private use, through non-transactional code, by the process that invoked T_i, constitutes privatization. Then, T_i is called privatizing transaction. While the privatization is not visible to all processes, inconsistencies may arise, given that for T_i's process the node is private but for other processes, the node is still seen as shared. Several solutions have been proposed for the privatization problem such as [159, 144, 147]. A system that provides *strong isolation* has the advantage of inherently also solving the privatization problem, because it inherently imposes synchronization between transactional and non-transactional code.

4.6 A Brief Presentation of TL2

TL2, aspects of which are used in this paper, has been introduced by Dice, Shalev and Shavit in 2006 [148]. The word-based version of the algorithm is used, where transactional reads and writes are to single memory words.

Main Features of TL2. The shared variables that a transaction reads form its *read set*, while the variables it updates form the *write set*. Read operations in TL2 are *invisible*, meaning that when a transaction reads a shared variable, there is no indication of the read to other transactions. Write operations are *deferred*, meaning that TL2 does not perform the updates as soon as it "encounters" the shared variables that it has to write to. Instead, the updates it has to perform are logged into a local list (also called *redo log*) and are applied to the shared memory only once the transaction is certain to commit. Read-only transactions in TL2 are considered efficient, because they don't need to maintain local copies of a read or write set and because they need no final read set validation in order to commit. To control transaction synchronization, TL2 employs locks and logical dates.

[1]Conversely, a memory area is made public when it goes from being exclusively accessible by one process to being accessible by several processes [162] . This is referred to as the *publication problem* and the consistency issues that arise are analogous.

Locks and Logical Date. A lock is associated with each shared variable. When a transaction attempts to commit it first has to obtain the locks of the variables of its write set, before it can update them. Furthermore, a transaction has to check the logical dates of the variables in its read set in order to ensure that the values it has read correspond to a consistent snapshot of shared memory. TL2 implements logical time as an integer counter denoted *GVC*. When a transaction starts it reads the current value of *GVC* into local variable, *rv*. When a transaction attempts to commit, it performs an increment-and-fetch on *GVC*, and stores the return value in local variable *wv* (which can be seen as a write version number or a version timestamp). Should the transaction commit, it will assign its *wv* as the new logical date of the shared variables in its write set. A transaction must abort if its read set is not valid. Its read set is valid if the logical date of every item in the set is less than the transaction's *rv* value. If, on the contrary, the logical date of a read set item is larger than the *rv* of the transaction, then a concurrent transaction has updated this item, invalidating the read.

4.7 Implementing Terminating Strong Isolation

A possible solution to the problem of ensuring isolation in the presence of non-transactional code consists in using locks: Each shared variable would then be associated with a lock and both transactions as well as non-transactional operations would have to access the lock before accessing the variable.

Locks are already used in TM algorithms - such as TL2 itself - where it is however assumed that shared memory is only accessed through transactions. The use of locks in a TM algorithm entails blocking and may even lead a process to starvation. However, it can be argued that these characteristics are acceptable, given that the programmer accepts the fact that a transaction has a duration and that it may even fail: The fact that there is always a possibility that a transaction will abort means that the eventuality of failure to complete can be considered a part of the transaction concept.

On the contrary, when it comes to single read or write accesses to a shared variable, a non-transactional operation is understood as an event that happens atomically and completes. Unfortunately strong isolation implemented with locks entails the blocking of non-transactional read and write operations and would not provide termination.

Given that this approach would be rather counter-intuitive for the programmer (as well as possibly detrimental for program efficiency), the algorithm presented in this section provides a solution for adding strong isolation which is not based on locks for the execution of non-transactional operations. This algorithm builds on the base of TM algorithm TL2 and extends it in order to account for non-transactional operations. While read and write operations that appear inside a transaction follow the original TL2 algorithm rather closely (cheap read only transactions, commit-time locking, write-back), the proposed algorithm specifies non-transactional read and write operations that are to be used by the programmer, substituting conventional shared memory read and write operations. TM with strong isolation has also been proposed in software [158, 161] in hardware [156], and has been suggested to be too costly [146]. This work differs from other implementations in that it is terminating and is implemented on top of a state-of-the-art STM in order to avoid too much extra cost.

4.7.1 Memory Set-up and Data Structures.

Memory Set-up. The underlying memory system is made up of atomic read/write registers. Moreover some of them can also be accessed by the the following two operations. The operation denoted Fetch&increment() atomically adds one to the register and returns its previous value. The operation denoted C&S() (for compare and swap) is a conditional write. C&S(x,a,b) writes b into x iff $x = a$. In that case it returns *true*. Otherwise it returns *false*.

The proposed algorithm assumes that the variables are of types and values that can be stored in a memory word. This assumption aids in the clarity of the algorithm description but it is also justified by the fact that the algorithm extends TL2, an algorithm that is designed to be word-based.

As in TL2, the variable *GVC* acts as global clock which is incremented by update transactions. Apart from a global notion of "time", there exists also a local one; each process maintains a local variable denoted *time*, which is used in order to keep track of when, with respect to the *GVC*, a non-transactional operation or a transaction was last performed by the process. This variable is then used during non-transactional operations to ensure the (strict) serialization of operations is not violated.

In TL2 a shared array of locks is maintained and each shared memory word is associated with a lock in this array by some function. Given this, a memory word directly contains the value of the variable that is stored in it. Instead, the algorithm presented here, uses a different memory set-up that does not require a lock array, but does require an extra level of indirection when loading and storing values in memory. Instead of storing the value of a variable directly to a memory word, each write operation on variable *var*, transactional or non-transactional, first creates an algorithm-specific structure that contains the new value of *var*, as well as necessary meta-data and second stores a pointer to this structure in the memory word. The memory set-up is illustrated in Fig. 4.2. Given the particular memory arrangement that the algorithm uses, pointers are used in order to load and store items from memory. [2]

T-record and NT-record. These algorithm-specific data structures are shared and can be of either two kinds, which will be referred to as T-records and NT-records. A T-record is created by a transactional write operation while an NT-record is created by a non-transactional write operation.

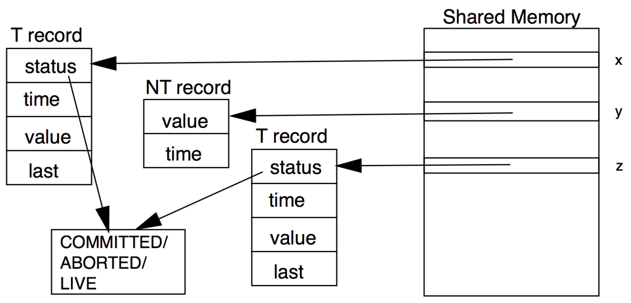


Figure 4.2: The memory set-up and the data structures that are used by the algorithm.

[2]The following notation is used. If pt is a pointer, $pt\downarrow$ is the object pointed to by pt. if aa is an object, $\uparrow aa$ is a pointer to aa. Hence $((\uparrow aa)\downarrow = aa$ and $\uparrow(pt\downarrow) = pt$.

New T-records are created during the transactional write operations. Then during the commit operation the pointer stored at *addr* is updated to point to this new T-record. During NT-write operations new NT-records are created and the pointer at *addr* is updated to point to the records.

When a read operation - be it transactional or non-transactional - accesses a shared variable it cannot know beforehand what type of record it will find. Therefore, it can be seen in the algorithm listings, that whenever a record is accessed, the operation checks its type, i.e., it checks whether it is a T-record or an NT-record (for example, line 209 in Fig. 4.3 contains such a check. A T-record is "of type T", while an NT-record is "of type NT").

T-record. A T-record is a structure containing the following fields.

status This field indicates the state of the transaction that created the T-record. The state can either be LIVE, COMMITTED or ABORTED. The state is initially set to LIVE and is not set to COMMITTED until during the commit operation when all locations of the transaction's write set have been set to point to the transaction's T-records and the transaction has validated its read set. Since a transaction can write to multiple locations, the *status* field does not directly store the state, instead it contains a pointer to a memory location containing the state for the transaction. Therefore the *status* field of each T-record created by the same transaction will point to the same location. This ensures that any change to the transaction's state is immediately recognized at each record.

time The *time* field of a T-record contains the value of the *GVC* at the moment the record was inserted to memory. This is similar to the logical dates of TL2.

value This field contains the value that is meant to be written to the chosen memory location.

last During the commit operation, locations are updated to point to the committing transaction's T-records, overwriting the previous value that was stored in this location. Failed validation or concurrent non-transactional operations may cause this transaction to abort after it updates some memory locations, but before it fully commits. Due to this, the previous value of the location needs to be available for future reads. Instead of rolling back old memory values, the *last* field of a T-record is used, storing the previous value of this location.

NT-record. An NT-record is a structure containing the following fields.

value This field contains the value that is meant to be written to the chosen memory location.

time As in the case of T-records, the *time* field of NT-records also stores the value of the *GVC* when the write took place.

Due to this different memory structure a shared lock array is no longer needed, instead of locking each location in the write set during the commit operation, this algorithm performs a compare and swap directly on each memory location changing the address to point to one of its T-records. After a successful compare and swap and before the transactions status has been set to COMMITTED or ABORTED, the transaction effectively owns the lock on this location. Like in TL2, any concurrent transaction that reads the location and sees that it is locked ($status =$ LIVE) will abort itself.

Transactional Read and Write Sets. Like TL2, read only transactions do not use read sets while update transactions do. The read set is made up of a set of tuples for each location read, $\langle addr, value \rangle$ where *addr* is the address of the location read and *value* is the value. The write set is also made up of tuples for each location written by the transaction, $\langle addr, item \rangle$ where *addr* is the location to be written and *item* is a T-record for this location.

4.7.1.1 Discussion.

One advantage of the TL2 algorithm is in its memory layout. This is because reads and writes happen directly to memory (without indirection) and the main amount of additional memory that is used is in the lock array. Unfortunately this algorithm breaks that and requires an additional level of indirection as well as additional memory per location. While garbage collection will be required for old T- and NT-records, here we assume automatic garbage collection such as that provided in Java, but additional solutions will be explored in future work. These additional requirements can be an acceptable trade-off given that they are only needed for memory that will be shared between transactions. In the appendix of this paper we present two variations of the algorithm that trade off different memory schemes for different costs to the transactional and non-transactional operations.

4.7.2 Description of the Algorithm.

The main goal of the algorithm is to provide strong isolation in such a way that the non-transactional operations are never blocked. In order to achieve this, the algorithm delegates most of its concurrency control and consistency checks to the transactional code. Non-transactional operations access and modify memory locations without waiting for concurrent transactions and it is mainly up to transactions accessing the same location to deal with ensuring safe concurrency. As a result, this algorithm gives high priority to non-transactional code.

4.7.3 Non-transactional Operations.

Algorithm-specific read and write operations shown in Fig. 4.3 must be used when a shared variable is accessed accessed outside of a transaction. This be done by hand or applied by a complier.

Non-transactional Read. The operation `non_transactional_read()` is used to read, when not in a transaction, the value stored at *addr*. The operation first dereferences the pointer stored at *addr* (line 208). If the item is a T-record that was created by a transaction which has not yet committed then the *value* field cannot be immediately be read as the transaction might still abort. Also if the current process has read (or written to) a value that is more recent then the transaction (meaning the process's *time* field is greater or equal to the T-records *time*, line 210) then the transaction must be directed to abort (line 211) so that opacity and strong isolation (containment specifically) is not violated. From a T-record with a transaction that is not committed, the value from the *last* field is stored to a local variable (line 213) and will be returned on operation completion. Otherwise the *value* field of the T- or NT-record is used (line 214).

Next the process local variable *time* is advanced to the maximal value among its current value and the logical date of the T- or NT-record whose value was read. Finally if *time* was set to ∞ on line 218 (meaning the T- or NT-record had yet to set its *time*), then it is updated to the

```
operation non_transactional_read($addr$) is
(208)  $tmp \leftarrow (\downarrow addr)$;
(209)  if ( $tmp$ is of type T $\wedge(\downarrow tmp.status) \neq$ COMMITTED )
(210)     then if ($tmp.time \leq time \wedge (\downarrow tmp.status) =$ LIVE)
(211)           then $C\&S(tmp.status$, LIVE, ABORTED) end if;
(212)        if (($\downarrow tmp.status) \neq$ COMMITTED)
(213)           then $value \leftarrow tmp.last$
(214)           else $value \leftarrow tmp.value$
(215)        end if;
(216)     else $value \leftarrow tmp.value$
(217)  end if;
(218)  $time \leftarrow \max(time, tmp.time)$
(219)  if ($time = \infty$) then $time = GCV$ end if;
(220)  return ($value$)
end operation.

operation non_transactional_write($addr, value$) is
(221)  allocate new variable $next\_write$ of type NT;
(222)  $next\_write \leftarrow (addr, value, \infty)$;
(223)  $addr \leftarrow (\uparrow next\_write)$
(224)  $time \leftarrow GVC$;
(225)  $next\_write.time \leftarrow time$;
end operation.
```

Figure 4.3: Non-transactional operations for reading and writing a variable.

GCV on line 219. The updated *time* value is used to prevent consistency violations. Once these book-keeping operations are finished, the local variable *value* is returned (line 220).

Non-transactional Write. The operation non_transactional_write() is used to write to a shared variable *var* by non-transactional code. The operation takes as input the address of the shared variable as well as the value to be written to it. This operation creates a new NT-record (line 221), fills in its fields (line 222) and changes the pointer stored in *addr* so that it references the new record it has created (line 223). Unlike update transactions, non-transactional writes do not increment the global clock variable *GCV*. Instead they just read *GCV* and set the NT-record's time value as well as the process local *time* to the value read (line 224 and 225). Since the *GCV* is not incremented, several NT-records might have the same *time* value as some transaction. When such a situation is recognized where a live transaction has the same time value as an NT-record the transaction must be aborted (if recognized during an NT-read operation, line 211) or perform read set validation (if during a transactional read operation, line 230 of Fig. 4.4). This is done in order to prevent consistency violations caused by the NT-writes not updating the *GCV*.

4.7.4 Transactional Read and Write Operations.

The transactional operations for performing reads and writes are presented in Fig. 4.4.

Transactional Read. The operation transactional_read() takes *addr* as input. It starts by checking whether the desired variable already exists in the transaction's write set, in which case the value stored there will be returned (line 226). If the variable is not contained in the write set, the pointer in *addr* is dereferenced (line 227) and set to *tmp*. Once this is detected to be a T- or NT-record some checks are then performed in order to ensure correctness.

In the case that *tmp* is a T-record the operation must check to see if the status of the transaction for this record is still LIVE and if it is the current transaction is aborted (line 236). This is similar to a transaction in TL2 aborting itself when a locked location is found. Next the T-record's *time* field is checked, and (similar to TL2) if it greater then the process's local *rv* value the transaction must abort (line 239) in order to prevent consistency violations. If this succeeds without aborting then the local variable *value* is set depending on the stats of the transaction that created the T-record (line 236-237).

In case *tmp* is an NT-record (line 228), the operation checks whether the value of the *time* field is greater or equal to the process local *rv* value. If it is, then this write has possibly occurred after the start of this transaction and there are several possibilities. In the case of an update transaction validation must be preformed, ensuring that none of the values it has read have been updated (line 230). In the case of a read only transaction, the transaction is aborted and restarted as an update transaction (line 231). It is restarted as an update transaction so that it has a read set that it can validate in case this situation occurs again. Finally local variable *value* is set to be the value of the *value* field of the *tmp* (line 233).

It should be noted that the reason why the checks are performed differently for NT-records and T-records is because the NT-write operations do not update the global clock value while update transaction do. This means that the checks must be more conservative in order to ensure correctness. If performing per value validation or restarting the transaction as an update transaction is found to be too expensive, a third possibility would be to just increment the global clock, then restart the transaction as normal.

Finally to finish the read operation, the $\langle addr, value \rangle$ is added to the read set if the transaction is an update transaction (line 241), and the value of the local variable *value* is returned.

```
operation transactional_read($addr$) is
(226) if $addr \in ws$ then return ($item.value$ from $addr$ in $ws$) end if;
(227) $tmp \leftarrow (\downarrow addr)$;
(228) if ($tmp$ is of type NT)
(229)    then if ($tmp.time >= rv$)
(230)           then if this is an update transaction then validate_by_value()
(231)                else abort() and restart as an update transaction end if;
(232)           end if;
(233)           $value \leftarrow tmp.value$;
(234)    else
(235)           if (($status \leftarrow (\downarrow tmp.status)) \neq$ COMMITTED )
(236)              then if ($status =$ LIVE) then abort() else $value \leftarrow tmp.last$ end if;
(237)              else $value \leftarrow tmp.value$
(238)           end if;
(239)           if ($tmp.time > rv$) then abort() end if;
(240) end if;
(241) if this is an update transaction then add $\langle addr, value \rangle$ to $rs$ end if;
(242) return ($value$)
end operation.

operation transactional_write($addr, value$) is
(243) if $addr \notin ws$
(244)    then allocate a new variable $item$ of type $T$;
(245)         $item \leftarrow (value, (\uparrow status), \infty)$; $ws \leftarrow ws \cup \langle addr, item \rangle$;
(246)    else set $item.value$ with $addr$ in $ws$ to $value$
(247) end if;
end operation.
```

Figure 4.4: Transactional operations for reading and writing a variable.

Transactional Write. The transactional_write() operation takes *addr* as input value, as well as the value to be written to *var*. As TL2, the algorithm performs commit-time updates of the variables it writes to. For this reason, the transactional write operation simply creates a T-record and fills in some of its fields (lines 244 - 245) and adds it to the write set. However, in the case that a T-record corresponding to *addr* was already present in the write set, the *value* field of the corresponding T-record is simply updated (line 246).

Begin and End of a Transaction The operations that begin and end a transaction are begin_transaction() and try_to_commit(), presented in Fig. 4.5. Local variables necessary for transaction execution are initialized by begin_transaction(). This includes *rv* which is set to *GCV* and, like in TL2, is used during transactional reads to ensure correctness, as well as *status* which is set to LIVE and the read and write sets which are initialized as empty sets. (lines 248-250).

```
operation begin_transaction() is
(248)   determine whether transaction is update transaction based on compiler/user input
(249)   $rv \leftarrow GVC$; Allocate new variable $status$;
(250)   $status \leftarrow$LIVE;  $ws \leftarrow \emptyset$; $rs \leftarrow \emptyset$
end operation.

operation try_to_commit() is
(251)   if ($ws = \emptyset$) then return (COMMITTED) end if;
(252)   for each ($\langle addr, item \rangle \in ws$) do
(253)      $tmp \leftarrow (\downarrow addr)$;
(254)      if ($tmp$ is of type $T \wedge (status \leftarrow (\downarrow tmp.status)) \neq$ COMMITTED )
(255)            then if ($status =$ LIVE) then abort() else $item.last \leftarrow tmp.last$ end if;
(256)            else $item.last \leftarrow tmp.value$
(257)      end if;
(258)      $item.time \leftarrow tmp.time$;
(259)      if ($\neg C\&S(addr, tmp, item)$) then abort() end if;
(260)   end for;
(261)   $time \leftarrow$ fetch&increment($GVC$); validate_by_value();
(262)   for each ($\langle addr, item \rangle \in ws$) do
(263)      $item.time \leftarrow time$;
(264)      if ($item \neq (\downarrow addr)$) then abort() end if;
(265)   end for;
(266)   if $C\&S(status$, LIVE, COMMITTED)
(267)      then return (COMMITTED)
(268)      else abort()
(269)   end if;
end operation.
```

Figure 4.5: Transaction begin/commit.

After performing all required read and write operations, a transaction tries to commit, using the operation try_to_commit(). Similar to TL2, a try_to_commit() operation starts by trivially committing if the transaction was a read-only one (line 251) while an update transaction must announce to concurrent operations what locations it will be updating (the items in the write set). However, the algorithm differs here from TL2, given that it is faced with concurrent non-transactional operations that do not rely on locks and never block. This implies that even after acquiring the locks for all items in its write set, a transaction could be "outrun" by a non-transactional operation that writes to one of those items causing the transaction to be required to abort in order to ensure correctness. As described previously, while TL2 locks items in its write set using a lock array, this algorithm compare and swaps pointers directly to the T-records in its write set (lines 252-260) while keeping a reference to the previous value. The previous value is stored in the T-record before the compare and swap is performed (lines 255-256) with a failed compare and swap resulting in the abort of the transaction. If while performing these compare and swaps the transaction notices that another LIVE transaction is updating this memory, it aborts itself (line 255). By using these T-records instead of locks concurrent operations have access to necessary metadata used to ensure correctness.

The operation then advances the *GVC*, taking the new value of the clock as the logical time for this transaction (line 261). Following this, the read set of the transaction is validated for correctness (line 261). Once validation has been performed the operation must ensure that non of its writes have been concurrently overwritten by non-transactional operations (lines 262-265) if so then the transaction must abort in order to (line 264) to ensure consistency. During this check the transaction updates the *time* value of its T-records to the transactions logical time (line 263) similar to the way TL2 stores time values in the lock array so that future operations will know the serialization of this transaction's updates.

Finally the transaction can mark its updates as valid by changing its *status* variable from LIVE to COMMITTED (line 266). This is done using a compare and swap as there could be a concurrent non-transactional operations trying to abort the transaction. If this succeeds then the transaction has successfully committed, otherwise it must abort and restart.

```
operation validate_by_value() is
(270)  $rv \leftarrow GVC$;
(271)  for each $\langle addr, value \rangle$ in $rs$ do
(272)     $tmp \leftarrow (\downarrow addr)$;
(273)     if ($tmp$ is of type T $\wedge tmp.status \neq$ COMMITTED)
(274)           then if ($tmp.status =$ LIVE $\wedge item \notin ws$) then abort() end if;
(275)              $new\_value \leftarrow tmp.last$;
(276)           else $new\_value \leftarrow tmp.value$
(277)     end if;
(278)     if $new\_value \neq value$ then abort() end if;
(279)  end for;
end operation.

operation abort() is
(280)  $status \leftarrow$ ABORTED;
(281)  the transaction is aborted and restarted
end operation.
```

Figure 4.6: Transactional helper operations.

Transactional Helping Operations. Apart from the basic operations for starting, committing, reading and writing, a transaction makes use of helper operations to perform aborts and validate the read set. Pseudo-code for this kind of helper operations is given in Fig. 4.6.

Operation validate_by_value() is an operation that performs validation of the read set of a transaction. Validation fails if any location in *rs* is currently being updated by another transaction (line 274) or has had its changed since it was first read by the transaction (line 278) otherwise it succeeds. The transaction is immediately aborted if validation fails (lines 274, 278). Before the validation is performed the local variable *rv* is updated to be the current value of *GVC* (line 270). This is done because if validation succeeds then transaction is valid at this time with a larger clock value possibly preventing future validations and aborts.

When a transaction is aborted in the present algorithm, the status of the current transaction is set to ABORTED (line 280) and it is immediately restarted as a new transaction.

4.8 Conclusion

This paper has presented an algorithm that achieves non-blocking strong isolation "on top of" a TM algorithm based on logical dates and locks, namely TL2. In the case of a conflict between a transactional and a non-transactional operation, this algorithm gives priority to the non-transactional operation, with the reasoning that while an eventual abort or restart is part of the specification of a transaction, this is not the case for a single shared read or write operation. Due to this priority mechanism, the proposed algorithm is particularly appropriate for environments in which processes do not rely heavily on the use of especially large transactions along with non-transactional write operations. In such environments, terminating strong isolation is provided for transactions, while conventional read and write operations execute with a small additional overhead.

4.9 Version of algorithm that does not use NT-records

This algorithm also provides wait-free NT read and write operations. The difference is that NT-records are not used. Instead NT values are read and written directly from memory. By doing this, memory allocations are not needed in NT writes and NT reads have one less level of indirection.

The cost of this is more frequent validations required in transactions when conflicts with NT writes occur. This algorithm is shown in Figs. 4.7-4.9.

```
operation non_transactional_read(addr) is
(282)  tmp ← (↓ addr);
(283)  if ( tmp is of type T )
(284)     then if (tmp.status = LIVE)
(285)              then C&S(tmp.status, LIVE, ABORTED)
(286)           end if;
(287)           if (tmp.status = ABORTED)
(288)              then value ← tmp.last
(289)              else value ← tmp.value
(290)           end if;
(291)     else value ← tmp
(292)  end if;
(293)  return (value)
end operation.

operation non_transactional_write(addr, value) is
(294)  addr ← (↑ unMark(value)) end operation.
```

Figure 4.7: Non-transactional operations for reading and writing a variable.

4.10 Version of algorithm with non-blocking NT-reads and blocking NT-writes

This algorithm allows wait-free NT read operations. The only change that is needed to the base TL2 algorithm is that when an item is locked it points to the write-set of the transaction, and that each transaction has a marker that is initialized as *LIVE* and is set to *COMMITTED* just before

```
operation transactional_read($addr$) is
(295) if $addr \in ws$ then return ($item.value$ from $addr$ in $ws$) end if;
(296) $tmp \leftarrow (\downarrow addr)$;
(297) if ($tmp$ is of type $T$ )
(298)    then if ($status$ = LIVE) then abort() end if;
(299)          if ($tmp.time > rv$) then abort() end if;
(300)          if ($status$ = COMMITTED)
(301)             then $value \leftarrow tmp.val$
(302)             else $value \leftarrow tmp.last$
(303)          end if;
(304)    else
      % Do validation to prevent abort due to a non-transactional write
(305)          $rv \leftarrow$ validate_by_value();
(306)          $value \leftarrow tmp$;
(307) end if;
(308) if this is an update transaction then add $value$ to $rs$ end if;
(309) return ($value$)
end operation.

operation transactional_write($addr, value$) is
(310) if $addr \notin ws$
(311)    then allocate a new variable $item$ of type $T$;
(312)          $item \leftarrow (addr, value, status, \infty)$; $ws \leftarrow ws \cup item$
(313)    else  set $item.value$ with $addr$ in $ws$ to $value$
(314) end if
end operation.
```

Figure 4.8: Transactional operations for reading and writing a variable.

the transaction starts performing write backs during the commit phase. The NT-read operation is shown in Fig. 4.10.

```
operation try_to_commit() is
(315)  if ($ws = \emptyset$) then return (COMMITTED) end if;
(316)  for each ($item \in ws$) do
(317)     $tmp \leftarrow (\downarrow addr)$;
(318)     if ($tmp$ is of type $T$)
(319)           then if (($status \leftarrow tmp.status$) = COMMITTED)
(320)                 then $item.last \leftarrow tmp.value$
(321)                 else if ($status$ = ABORTED) then $item.last \leftarrow tmp.last$
(322)                 else abort()
(323)              end if;
(324)           else $item.last \leftarrow tmp$
(325)     end if;
(326)     if ($\neg C\&S(item.addr, tmp, item)$) then abort() end if;
(327)  end for;
(328)  $time \leftarrow$ fetch&increment($GVC$);
(329)  validate_by_value();
       % Ensure the writes haven't been overwritten by non-transactional writes
(330)  for each ($item \in ws$) do
(331)     if ($item \neq (\downarrow item.addr)$) then abort() end if
(332)     $item.time \leftarrow time$;
(333)  end for;
(334)  if ($C\&S(status$, LIVE, COMMITTED))
(335)     then return (COMMITTED)
(336)     else abort()
(337)  end if;
end operation.
```

Figure 4.9: Transaction commit.

```
operation non_transactional_read($addr$) is
(338)  $lock \leftarrow$ load_lock($addr$);
(339)  $value \leftarrow (\downarrow addr)$;
(340)  if ( $lock$ is locked $\wedge tmp.status$ = COMMITTED $\wedge addr \in lock.ws$)
(341)     then $value \leftarrow item.value$ from $addr$ in $lock.ws$
(342)  end if;
(343)  return ($value$)
end operation.

operation non_transactional_write($addr, value$) is
(344)  Perform a transactional begin/write/commit operation
end operation.
```

Figure 4.10: Non-transactional operations for reading and writing a variable.

Part II

Concurrent Data Structures

Chapter 5

Introduction

Chapter 6

Contention Friendly Data Structures

6.1 Non-Contention Friendly Data Structures

6.2 The Contention Friendly Methodology

6.2.1 In Transactional Memory

6.3 Introduction

Multicore architectures are changing the way we write programs. Not only are all computational devices turning multicore thus becoming inherently concurrent, but tomorrow's multicore will embed a larger amount of simplified cores to better handle energy while proposing higher performance, a technology usually called *manycore* [127]. Programmers must thus change their habits to design new concurrent data structures that can be bottlenecks in modern every day applications.

The big-oh complexity, which indicates the worst-case amount of converging steps necessary to complete an access, used to prevail in the choice of a particular data structure algorithm running in a sequential context or with limited concurrency. Yet contention has now become an even more important factor of performance drops in today's multicore systems. For example, some concurrent data structures are even so contended that they cannot perform better than bare sequential code, and exploiting additional cores simply make the problem worse [140]. In response to such contention, researchers seek relaxed abstractions, i.e., alternative abstractions offering weaker guarantees, whose performance remains acceptable when their data structure implementation is placed in a highly concurrent context. This is typically the case for the queue of the Intel® TBB[1] that is not FIFO under multiple producers/consumers and for the quiescently consistent stack that is not LIFO in the presence of concurrency [140].

To better illustrate how contention can counterbalance the big-oh complexity in today's multi-/many-cores, Figure 6.1 depicts the performance of a 48-core machine running the same set based experiment on a concurrent linked list, with $O(n)$ complexity, and on a concurrent skip list, with $O(\log_2 n)$ complexity. A skip list, in short, is a structure that diminishes the complexity of a linked list by being a sort of linked list whose nodes may have additional shortcuts pointing towards other nodes located further in the list [139]. In this experiment, 48 threads run insert/delete/contains accesses with an increasing proportion of update accesses over read-only

[1] Intel® Threading Building Blocks (TBB) **http://threadingbuildingblocks.org**http://threadingbuildingblocks.org.

ones on each of these two structures initialized with 512 elements.[2] To obtain the corresponding concurrent data structures used in the experiments, we simply encapsulated the sequential code of each access into an elastic transaction [131]. Interestingly, above 20% updates the concurrent linked list is more efficient than the concurrent skip list—this is shown by the negative values of the speedup-1. The reason is that the linked list updates are localized, that is, each of them only affects a constant number of nodes, typically the predecessor of the removed or of the newly inserted node. By contrast, the skip list updates may affect up to a logarithmic amount of predecessors for each removed of newly inserted node, producing additional contention.

This result is unsurprising as Herb Sutter noticed that linked list could better tolerate contention than balanced trees for similar reasons [124], yet it is interesting to observe experimentally that only 20% updates make a linear complexity data structure better suited than a logarithmic complexity data structure on nowadays' multicore machines.

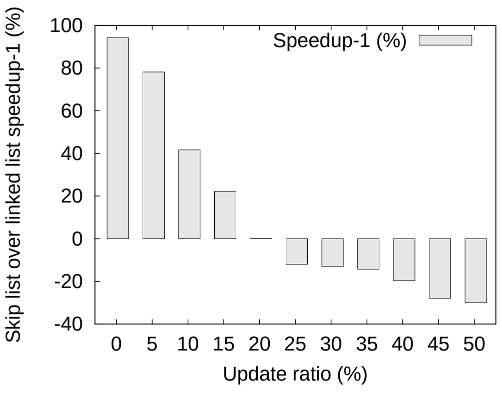


Figure 6.1: Impact of contention on the performance of two 512-sized data structures (with 48 cores running with an increasing update ratio)

In the light of the impact of contention on performance, we propose the *Contention-Friendly (CF)* methodology as a methodology to design new data structures that accommodate contention of modern multi-/many-core machines without relaxing the correctness of the abstractions. To this end, we argue for a genuine decoupling of each access into an eager abstract access and a lazy structural adaptation. The abstract access consists in modifying the abstraction by minimizing the impact on the structure itself and aims at returning as soon as possible for the sake of responsiveness. The structural adaptation, which can be deferred until later, aims at adapting the structure to these changes by re-arranging elements or garbage collecting deleted ones.

We illustrate the CF methodology by designing three data structures with locks, universal primitives, and transactions: a skip list, a binary search tree and a hash table. As for the skip list, the aforementioned decoupling translates into splitting a node insertion into the insertion phase at the bottom level of a skip list and the structural adaptation responsible for updating pointers at its higher levels, or into splitting a node removal into a logical deletion marking phase and its physical removal and garbage collection. Similarly, the decoupling of the binary tree accesses consists in inserting or logically removing a node prior to rebalancing and/or garbage collecting. Finally, the hash table decoupling lies in inserting/deleting eagerly and resizing the structure lazily.

Our Java implementation of the resulting data structures indicates that our methodology leads to good performance on today's multicore machines. In particular using a micro benchmark, on a 64-way Niagara 2 machine our lock-based CF binary search tree improves the performance of the most recent Java lock-based binary search tree implementation [97] by up to 2.2×, our lock-based CF skip list improves the performance of Doug Lea's concurrent skip list adaptation of Harris and Michael algorithms [134, 137] by up to 1.3×, and our lock-free hash table outperforms by up to 1.2× the JDK hash table, which is widely distributed in the `java.util.concurrent` package. Finally, we show that state-of-the-art software transactional memories execute 1.5× faster on average when the data structures are contention-friendly.

[2]More precisely, this experiment was performed on a 4 × 12-core AMD Opteron machine running at 2.1GHz and 32 GB of memory, each point is averaged over 5 runs of 5 seconds each, removes and inserts accesses are triggered with the same probability to keep the size expectation constant and removes/inserts that do not update the data structure are considered read-only accesses.

Section 8.5 describes the related work. Section 8.6 depicts the CF methodology, Section 8.7 illustrates it on three data structures. Section 8.8 presents the experimental results and Section 8.9 concludes. The companion appendix comprises the pseudo-code and correctness proofs of our CF algorithms, as well as additional experimentations using transaction-based variants of our CF algorithms and a discussion.

6.4 Related Work

Various complexity metrics exist to evaluate data structures efficiency on a given workloads. From a theoretical point of view, the big-oh notation helps to derive data structures whose access step complexity is proportional to the total number of elements. Typically, balanced trees have a logarithmic big-oh access complexity whereas non-overloaded hash tables have a constant big-oh access complexity. This big-oh complexity does not capture the cost of contention and theoretical models have been explored to remedy this issue [100]. Unfortunately, there are not enough evaluations of this impact in practice.

From a more pragmatic point of view, the locality of data both in terms of space (i.e., the promiscuity of data stored in memory) and time (i.e., the closeness of the points in time at which they are accesses) has been an important metric of consideration when implementing data structures in cache-coherent systems. Cache-aware and cache-oblivious data structures try to exploit locality to maximize the chance of cache hits. While the former data structures rely on some tunable parameter that can accommodate the targeted platform like the Judy array[3], the later aims at being more portable by flattening cleverly structural nodes into memory [105], for example the tree algorithm of van Emde Boas et al. [125]. Both approaches are tied to cache-coherent machines but do not accommodate upcoming many-core platforms whose cache-coherence is either limited [126] or absent [115].

The decoupling of the update and rebalancing was vastly explored in the context of trees [106, 112, 138, 119, 114, 96, 94, 128] but this idea was not generalized to other search structures. The decoupling of the removals in logical and physical phases was originally studied in transactional systems [117] and later applied to various lock-free data structures including linked lists [134], hash tables [137], skip lists [133, 132] and binary search trees [101] but insertions in these data structures were not decoupled. The contention friendly methodology generalizes these decoupling into an eager abstract access and a lazy structural adaptation that benefit both insertions and removals.

Our methodology is independent from the synchronization primitive used but lies essentially in splitting accesses into an eager abstract access and a lazy structural adaptation. Although we focus essentially on lock-based data structures, we also evaluate the benefit of various transactional memory algorithms when running our contention-friendly data structures. We have already illustrated the benefit of decoupling accesses into separate transactions in [128] on a C-based binary search tree. In such optimistic executions, this decoupling translated into avoiding a conflict with a rotation from rolling back the preceding insertion/removal. Here we generalize our previous work by showing how a similar decoupling can benefit pessimistic execution and various search structures and we compare our results to existing Java concurrent structures. Previous investigations on improving the performance of transaction-based data structures focused exclusively on the improvement of the transaction algorithm. Some of these investigations

[3]`http://judy.sourceforge.net`

Data structure	Invariant	Abstract modifications	Structural adaptations
Hash tables	constant load factor (i.e., $\#\mathsf{nodes}/\#\mathsf{buckets} = O(1)$)	key-value pair insertion logical deletion	adding buckets and rehashing physical deletion + rehashing
Search trees	balance (i.e., shortest route to leaf $\approxeq$ longest route to leaf)	node insertion logical deletion	rotation physical deletion + rotation
Skip lists	node distribution per level (i.e., $\Pr[level_i = j] = 2^{O(j)}$)	horizontal insertion logical deletion	vertical insertion + increasing toplevel physical removal + decreasing toplevel

Table 6.1: Decoupling example of existing data structure accesses into an abstract modification and a structural adaptation

led to the development of novel transaction models based on abstract locks to ignore low level conflicts [118, 108], or elastic transactions [131].

Finally, Shavit suggests to relax data structure guarantees in the light of the new multicore context [140]. A stack algorithm and several relaxations to this algorithm are presented to support concurrency. The objective as well as the means to achieve it are quite different from ours. First, the problem raised by placing the stack into the multicore context is that performance drops below the sequential stack performance, and the goal is to diminish contention to limit this concurrency drawback. By contrast, we focus on deriving alternative data structures that are more scalable than highly concurrent ones, hence leveraging multi-/many-cores. Second, the goal of limiting contention induced by multiple cores is achieved by relaxing consistency. In the stack example, this relaxation boils down to replacing linearizability by quiescent consistency, guaranteeing that the last-in-first-out policy of an access is only with respect to preceding calls when no other accesses execute concurrently. Conversely, the contention friendly methodology aims at replacing existing data structures without relaxing their abstraction consistency: all accesses remain linearizable.

6.5 The CF Methodology at a Glance

In this section, we give an overview of the Contention-Friendly (CF) methodology by describing how to write contention-friendly data structures.

The CF methodology aims at modifying the implementation of existing data structures using two simple rules without relaxing their correctness. The correctness criterion ensured here is linearizability [135]. The data structures considered are *search structures* because they organize a set of items referred to as *elements* in a way that allows to retrieve the unique expected position of an element given its value. The typical abstraction implemented by such structures is a collection of elements that can be specialized into various sub-abstractions like a set (without duplicates) or a dictionary (that maps each element to some value). We consider *insert*, *delete* and *contains* operations that respectively inserts a new node associated to a given value, removes the node associated to a given value or leaves the structure unchanged if no such node is present, and returns the node associated to a given value or $\perp$ if such a node is absent. Both inserts and deletes are considered *updates*, even though they may not modify the structure.

The key rule of the methodology is to decouple each update into an *eager abstract modification* and a *lazy structural adaptation*. The secondary rule is to make the removal of nodes selective and tentatively affect the less loaded nodes of the data structure. These rules induce slight changes to the original data structures as summarized in Table 6.1, that result in a corresponding data structure that we denote using the *contention-friendly* adjective to differentiate them from their original counterpart.

6.5.1 Eager abstract modification

Existing search structures rely on strict invariants (cf. Table 6.1) to guarantee their big-oh complexity, hence each time the structure gets updated, the invariant is checked and the structure is accordingly adapted instantaneously. While the update may affect a small sub-part of the abstraction, its associated restructuring is a global modification that conflict potentially with any concurrent update, thus increasing contention.

The CF methodology aims at minimizing such contention by returning eagerly the modifications of the update operation that makes the changes to the abstraction visible. By returning eagerly, each individual process can move on to the next operation prior to adapting the structure. It is noteworthy that executing multiple abstract modifications without adapting the structure does no longer guarantee the big-oh step complexity of the accesses, yet such complexity may not be the predominant factor in contended execution as we reported in the Introduction.

A second advantage is that removing the structural adaption from the abstract modification makes the cost of each operation more predictable. All operations share similar cost and create the same amount of contention. More importantly the completion of the abstract operation does not depend on the structural adaptation (like they do in existing algorithms) so the structural adaptation can be performed differently, using and depending on global information.

The skip list example. A traditional skip list picks a level for each node when they are inserted based on some pseudo-random function. The aim of this function is to distribute the levels so that operations have an average cost of $O(\log n)$. In certain workloads this can be preferred over trees due to the assumption that rotations are more costly. When a node is inserted in the contention-friendly skip list it has a level of one, which is all that is needed to ensure the correctness of the abstraction.

As an example, assume we aim at inserting an element with value 12 in a skip list. Our insertion consists in an abstract modification that updates only the bottom most level by inserting the new node as if its level was the lowest one leading to Figure 8.1(a) where dashed arrows indicate the freshly modified pointers. We defer the process of linking this same node at higher levels, to diminish the probability of having this insertion conflict with a traversing operation.

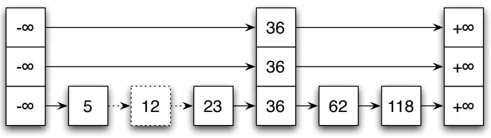


Figure 6.2: Inserting horizontally in the skip list

6.5.2 Lazy structural adaptation

The purpose of decoupling the structural adaptation from the preceding abstract modification is to enable its postponing (by, for example, dedicating a separate thread to this task), hence the term "lazy" structural adaptation. The main intuition here is that this structural adaptation is intend to ensure the big-oh complexity rater than to ensure correctness of the state of the abstraction. Hence, the linearization point belongs to the execution of the abstract modification and not the structural adaptation and postponing the structural adaptation does not change the effectiveness of operations. The visible modification applied to the abstraction (and the structure) during the abstract modification guarantees that any further operation applying to the same structure will observe the changes. This helps ensuring that all operations are linearizable in that real-time precedence is satisfied. In Appendix 6.11 we show that our structures implement a linearizable abstraction.

This postponing has several advantages whose prominent one is to enable merging of multiple adaptations in one simplified step. Although the structural adaptation might be executed in a distributed fashion, by each individual updater threads, one can consider centralizing it at one dedicated thread. Since these data structures are designed for architectures that use many cores performing the structural adaptation on a dedicated single separate thread, takes advantage of hardware that might otherwise be left idle. Only one adaptation might be necessary for several abstract modifications and minimizing the number of adaptations decreases accordingly the induced contention. Furthermore, several adaptations can compensate each other as two restructuring can lead to identity. For example, a left rotation executing before a right rotation at the same node may lead back to the initial state and executing the left rotation lazily makes it possible to identify that executing these rotations is useless.

The skip list example. As explained in the previous example the insertion executes in two steps. Once the horizontal insertion of node 12, depicted in Figure 8.1(a), is complete, a restructuring is necessary to ensure the logarithmic complexity of further accesses.

A separate structural adaptation step is accordingly raised to increase the node level appropriately. The insertion at higher levels of the skip list is executed as a separate step, which guarantees eventually a good distribution of nodes among levels as depicted in Figure 8.1(b). This decoupling allows higher concurrency by splitting one atomic operation into two atomic operations.

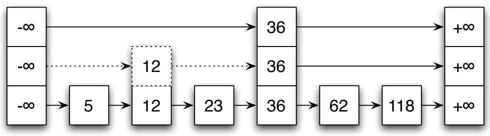


Figure 6.3: Adapting vertically the skip list structure

6.5.3 Selective removal

In addition to decoupling level adjustments, we do selective removals. A node that is deleted is not removed instantaneously, instead it is marked as deleted. The structural adaptation then selects cleverly nodes that are suitable for removal, i.e., whose removal would not induce high contention. This is important because removals may be expensive. Removing a frequently accessed node requires locking or invalidating a larger portion of the structure. Removing such a node is likely to cause much more contention than removing a less frequently accessed one. In order to prevent this, only nodes that are marked as deleted and have a level of 1 (in the skip list) or a single or no children (in the tree) are removed. This leads to less contention, but also means that certain nodes that are marked as deleted will not be removed. In the tree it has already been observed that only removing such nodes [128],[97] results in a similar sized structure as existing algorithms. In the skip list the level of a node is calculated in such a way that after a structural adaptation is performed less than half the nodes (in the worst case) in the list will be marked as deleted. In practice this number is observed to be much smaller.

The skip list example. Let us look at a specific example with the skip list. On the one hand, a removal of a node with a high level, say the one with value 36 in Figure 8.1(b), would typically induce more contention than the removal of a node with a lower level, say the one with value 62 spanning a single level. The reason is twofold. First removing a node spanning ℓ levels boils down to updating ℓ pointers which increase the probability of conflict with a concurrent operation accessing the same pointers, hence removing node with value 36 requires to update 3 pointers while node with value 63 requires to update a single pointer. Second, the organization of the skip list implies that higher level pointers are more likely accessed by any operation, hence the removal of 36 typically conflicts with every operation concurrently traversing this structure (because all these operations would follow the topmost left pointer) whereas the single next pointer of 62 is unlikely accessed by concurrent traversals. Removing a tall node such as 36 would also mean that in order to keep the logarithmic complexity of the traversals a node would have to take its place at an equivalent height.

6.5.4 Avoiding contention during traversal

Each abstract operation (*contains*, *insert*, *delete*) of a tree or a skip list is expected to traverse $O(\log n)$ nodes. Given that the traversal is the longest part of the operation, the CF algorithms try to avoid as often as possible producing contention. Concurrent data structures often require more complex synchronization operations during traversal (not including the updates done after the traversal). For example, locking nodes in a tree helps ensure that the traversal remains on track during a concurrent rotation [97], using compare-and-swap operations during traversal helps the raising and lowering of levels of a concurrent *insert*/*delete* in a lock-free skip list [133], or using optimistic strategy helps at the risk of having to restart [109, 110].

Usually these synchronization operations are required due to structural adaptations and the CF algorithms structural adapt differently to especially so that operations can avoid using locks or synchronization operations during traversal.

6.6 Putting the CF Methodology to Work

Here we present how we apply the contention-friendly (CF) methodology to three data structures. For further detail on the algorithms and correctness proofs please refer to Appendix 6.10 and Appendix 6.11, respectively.

6.6.1 CF Skip list

The CF skip list is made up of several levels of linked lists, with the bottom level being a doubly linked list. Each node on the bottom level contains the following fields: A key k, a *next* and *prev* node pointers, a lock field, a *del* flag indicating if the node has been marked deleted, and a *rem* flag indicating if the node has been physically removed. The algorithm presented here is lock-based, however, we have derived a transaction-based version (cf. Appendix 6.9).

Abstract operations. The goal of these CF algorithms is for the abstract operations to encounter and produce as little contention as possible. In particular, it boils down to setting the nodes *del* flag to true to delete a node as well as linking a new node to the bottom list level to insert it. These modifications are necessary to guarantee that linearizability, with all other structural adaptations being saved for later execution. For the sake of safety, the abstract insertion acquires a lock on the predecessor node of the to-be-inserted node whereas the abstract deletion acquires a lock on the to-be-marked node. The lock is immediately released after the insertion or deletion completes.

No locks are acquired during the traversal, inducing no contention. More precisely, while traversing upper levels the operation will move forward in the list using the *next* pointer until it encounters a node with a larger key than the one being search for at which point it will move down a level, similarly to a bare sequential implementation would do. At the bottom level the traversal may end up on a node that is physically removed due to a concurrent structural adaptation *remove* operation, in this case it travels backwards in the list following the *prev* pointer until it arrives at a node that has not yet been removed.

Structural adaptation. The first task of the structural adaptation is to remove nodes marked as deleted who have a height of 1. In order to prevent conflicts with concurrent abstract operations the node n to be removed and its predecessor in the list ($n.prev$) are locked. The prior nodes *next* pointer ($n.prev.next$) is then modified so that it points to the next node ($n.next$), and the next node's *prev* pointer ($n.next.prev$) is then modified to point to the previous node ($n.prev$). Finally the n's *rem* flag is set to true and the locks are released.

The structural adaptation must also modify the level of nodes in order to ensure the $O(\log n)$ expected traversal time. Since neither removals nor insertions are done as they are in traditional skip lists, calculating the height of a node must also be achieved differently. Existing algorithms call a random function to calculate the heights of nodes, but if this same function was used here the structure would end up with excessive tall nodes.

When choosing the heights it is important to consider that the fundamental structure of a skip list is not designed to be perfectly balanced but rather probabilistically balanced. Consider a perfectly balanced skip list. The node in the very middle of the list would be the tallest node and the nodes just to the right and left of this node would be nodes with height 1. Now if a couple new nodes are inserted at the very end of the list then to re-balance the skip list the node that was previously the tallest node would now be shrunk to a level of 1, and one of its neighboring nodes which previously had height of 1 would become the tallest node. Instead a scheme of approximately balanced is more fitting for the skip list (as this is what the existing algorithm's random functions do).

By contrast, the CF skip list deterministically adjusts the level of nodes. From the bottom level going upwards, it traverses the entire list of the level, and each time it observes that 3 consecutive nodes whose height equals this level, it raises the level of the second of this node (the one in the middle) by 1. Such a technique approximates the targeted number of nodes present at each level, balancing the structure. Doing this is similar to the original intuition of the skip list, there is no frequent re-balancing going on, tall nodes will stay tall nodes. Less modification of the taller nodes also means less contention at the frequently traversed locations of the structure.

Given that the number of nodes in the list might also shrink the height of nodes might also be lowered. When the height of the tallest node is greater than some threshold (usually when the height is greater than the log of the total number of nodes in the list) the entire bottom index level of the skip list is simply removed by modifying the *down* pointers of the level above. Doing this avoids constant modification of the taller nodes and ensures there are not too many marked deleted nodes left in the list.

6.6.2 CF Tree

The CF tree is a binary search tree. Each of its nodes contains the following fields: a key k, pointers l and r to the left and right children nodes, a lock field, a *del* flag indicating if the node has been marked deleted, and a *rem* flag indicating if the node has been physically removed. As for the CF skip list, the CF tree algorithm presented here is lock-based but we also derived a transaction-based variant of it.

Abstract operations. Similarly to the CF skip list operations the *insert* and *delete* operations must acquire a lock on the node they modify. A *delete* operation sets the node's *del* flag to `true` while an *insert* operation allocates a new node and modifies the parent's child pointer to point to it.

The traversal is performed without locks. At each node the traversal travels to the right child if the node's key is larger than k, otherwise it travels to the left child. Since locks are not used, the traversal might get caught during a concurrent removal or rotation, but the structural adaptation is done in such a way that the traversal can continue safely following the child pointers.

Structural adaptation. The structural adaptation is in charge of removing marked deleted nodes that have at most one non-$\perp$ child pointer. Removals are done by first locking the node n to be deleted and its parent. The parent's child pointer is then modified so that it points to n's non-$\perp$ child (if any). Next n's child pointers are modified so that they point upwards to it's parent node allowing concurrent traversal that arrived on this node a safe path back to the tree. Finally n's *rem* flag is set to `true` and the locks are released.

The structural adaptation must also perform rotations in order to ensure the tree is balanced so that traversal can be done in $O(\log n)$ time. Methods for performing localized rotation operations in the binary trees have already been examined and proposed in several works such as [128, 97]. The main concept used here is to propagate the balance information from a leaf to the root. A leaf is known to have height of 0 for their left and right children. This information is then propagated upwards by sending the height of the child to the parent where the value is then increased by 1. Local rotations are performed depending on this information and result eventually in a balanced tree.

In order to avoid using locks and aborts/rollbacks during traversals, rotations are performed differently than traditional rotations. Before performing the rotation the parent node and its child node that will be rotated are locked in order to prevent conflicts with concurrent *insert* and *delete* operations. In a traditional rotation there is one node n that is rotated downwards and one node (one of n's children) that is rotated upwards. A traversal preempted on the node rotated downwards (n in this case) is then in danger of being set off track and missing the node it is searching for. The rotations performed in the CF algorithm avoid this by not actually rotating n at all, meaning that after the rotation n still has a pointer to the node that is rotated upwards allowing traversals to continue safely. Instead a new node takes n place in the structure. This new node is set to have the same values and pointers as n would if a rotation was performed as normal. After the rotation, the node n has its *rem* flag set to `true` and, finally, the locks are released.

6.6.3 CF Hash table

The CF hash table contains an array of pointers with each location pointing to $\perp$ or to a list of nodes. Each node contains the following fields. A key k, and a *next* pointer pointing to the next node in the list. This algorithm is lock-free (relying on `compare-and-swap` for synchronization) but we derived a transaction-based variant of it (cf. Appendix 6.9).

Abstract operations. Given that the traversal for the *contains*, *insert*, *delete* operations has complexity $O(1)$ and not $O(\log n)$ the hash table operations are performed slightly differently. In fact, the shortness of the hash table operations brings two main differences to the algorithm.

First physical removals are done from within the *delete* operation. This is because the contention caused by removing the node will only be with other nodes of that bucket which are expected to be $O(1)$.

Second the algorithm is made lock-free because given the short operations, a cache miss caused by loading a lock could be relatively costly. Other implementations might avoid this by using coarser grained locks, like lock-striping, but this can cause contention on the lock(s). Instead we use a lock-free implementation where each operation only uses (at most) a single synchronization operation, which is a compare-and-swap on the given bucket pointer.

For the sake of linearizability of operations the compare-and-swap always happens at the same location (on the bucket pointer) and the *next* pointer of list elements is never modified after node creation. An *insert* will compare-and-swap a new node as the first element of the list, while a *delete* will remove a node by creating a new list that does not contain the node and compare-and-swap this new list to the bucket. If the compare-and-swap fails due to a concurrent operation then the operation retries from the beginning.

Structural adaptation. The structural adaptation must ensure the $O(1)$ cost of *contains*, *insert*, *delete* operations. This is done by rehashing and resizing the table which first traverses the table counting the number of nodes. If the number of nodes is greater than some threshold (usually a fraction of the number of buckets in the table) then a rehash is performed and the size of the table is increased by a size of the power of 2.

The rehash is performed one bucket at a time allowing concurrent operations on other buckets. At each bucket the list of nodes is copied and placed into two new lists added to the corresponding buckets of the new table. Next a compare-and-swap is performed at the bucket of the old table replacing the list there with a dummy node. If the compare-and-swap fails then the rehash operation is retried for this bucket. Any abstract operation that encounters a dummy node then knows that the bucket has been rehashed so it uses the new table for the operation.

6.7 Evaluation

We evaluate the CF methodology using a micro benchmark by comparing our CF data structures to three Java state-of-the-art concurrent data structure implementations:

- Non-CF hash table: the widely deployed ConcurrentHashMap of the java.util.concurrent package,
- Non-CF binary tree: the most recent lock-based binary search tree [97] we are aware of, and
- Non-CF skip list: the Doug Lea's ConcurrentSkipListMap relying on Harris and Michael algorithms [134, 137].

All CF data structure implementations use a separate thread in addition to the application threads that constantly adapts the structure to compensate the effect of preceding abstract modifications. We use an UltraSPARC T2 with 8 cores running up to 8 hardware threads each, comprising 64 hardware threads in total. For each run we averaged the number of executed operations per microsecond over 5 runs of 5 seconds. Thread counts are $1, 2, 4, 8, 16, 24, 32, 40, 48, 56$ and 64 and the five runs execute successively as part the same JVM for the sake of warmup. We used Java SE 1.6.0 12-ea in server mode and HotSpot JVM 11.2-b01.

Figure 6.4 depicts the tolerance to contention of the various data structures. More precisely, it indicates the slowdown of each data structure under contention as the normalized ratio of its performance with non-null update ratios over its performance without updates. The slowdown of non-CF tree and skip list always more significant than the one of their CF counterpart, indicating that the CF is more tolerant to contention.

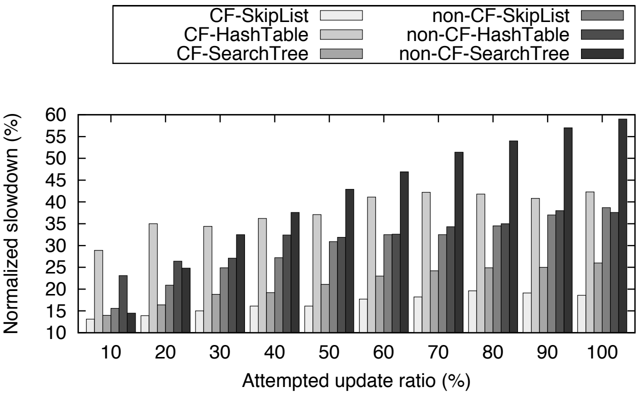


Figure 6.4: Tolerance to contention of Contention-Friendly (CF) and non-CF data structures (performance slowdown with respect to 0% updates)

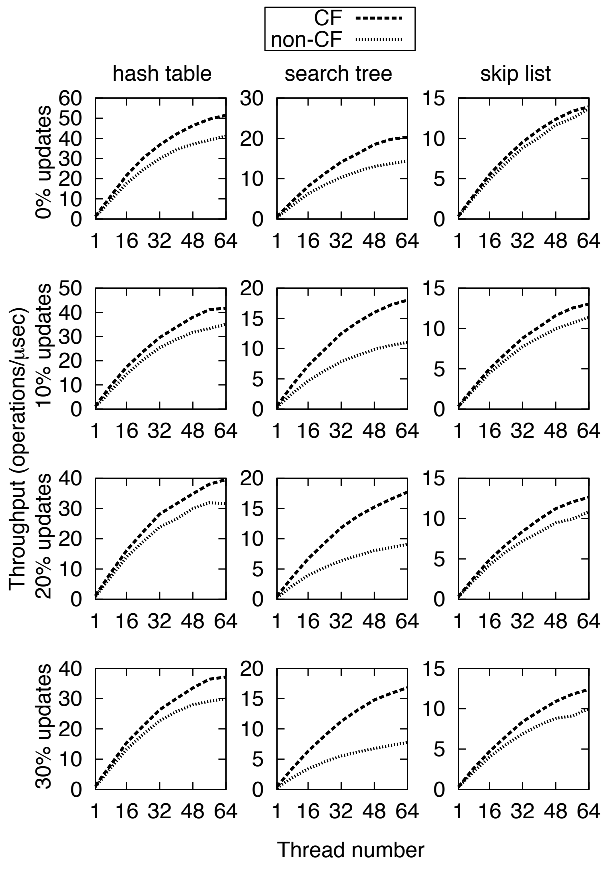
Interestingly, the slowdown of the CF hash table is higher than the one of the non-CF hash table at low levels of contention but becomes similar at high contention levels. As shown later, our CF hash table is actually very efficient on read-only workload whereas the ConcurrentHashMap relies on lock-stripes whose segments have to be loaded even on read-only workloads. This explains why CF performance drops as soon as contention appears, however, the CF hash table tolerates the contention increase as the slowdown remains almost constant, as opposed to the non-CF hash table. We played with the number of segments and we observed better scalability with more segments but lower read-only overhead with a single one. We chose 64 segments which makes threads fetch multiple segments from memory before finding them in their cache. Another advantage of using the CF hash table is not having to worry about such segments.

Figure 6.5 compares the performance of state-of-the-art data structures against performance of our CF data structures with 2^{14} (left) and 2^{16} elements (right) and on a read-only workload (top) and workloads comprising up to 30% updates (bottom). While all data structures scale well with the number of threads, the state-of-the-art data structures are slower than their contention-friendly counterparts in all the various settings. In particular, the CF hash table, skip list, search tree are respectively up to $1.2\times$, $1.3\times$, $2.2\times$ faster than their non-CF counterparts.

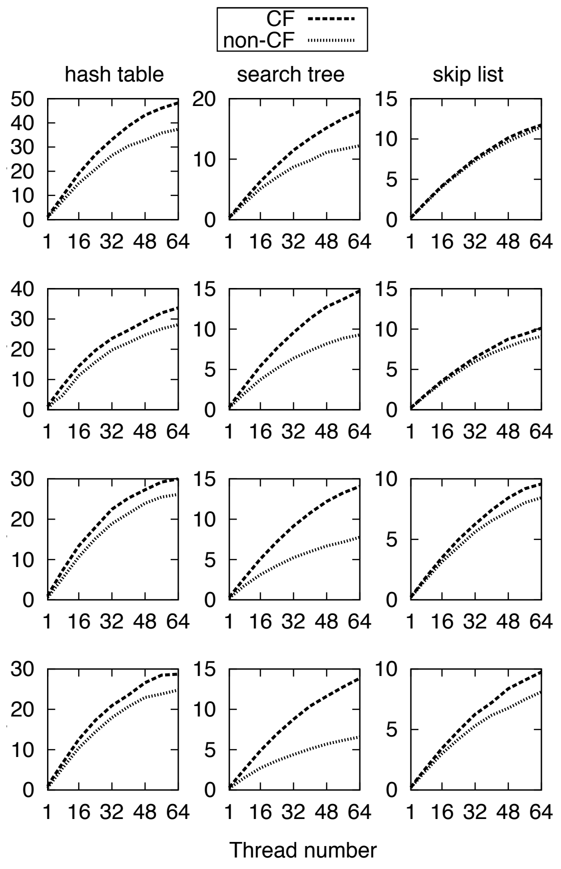
Finally Appendix 6.9 shows that our adaptation of these data structures to three transactional memory algorithms allows a performance benefit of $1.5\times$ on average.

6.8 Conclusion

Multicore programming brings new challenges, like contention, that programmers have to anticipate when developing novel applications. Programmers must now give up concentrating on the big-oh complexity and should rather think in terms of contention overhead. We explored the methodology of designing contention-friendly data structures, keeping in mind that contention will be a predominant cause of performance loss in tomorrow's architectures. This simple methodology led to a novel Java package of concurrent data structures more efficient than



(a) 2^{14} elements



(b) 2^{16} elements

Figure 6.5: Performance of the Contention-Friendly (CF) and non-CF data structures

the best implementations we could find. We plan to extend it with additional contention-friendly data structures.

6.9 Transactional Contention-Friendly Algorithms

The concept of splitting the abstract operation and the structural modification to create contention friendly data structures does not only apply to lock based or lock-free implementations. It can also be applied to data structures implementations using transactional memory. Our previous work has studied this problem when specifically looking at trees [128].

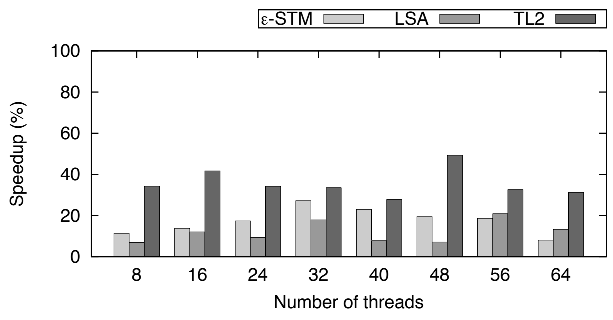
We now present the experimental results with three existing transactional memory implementations: $\mathscr{E}$-STM [131], LSA [122] and TL2 [99] using the Deuce Java bytecode instrumentation framework [113]. The experimental settings are the same as for other experiments, except that we evaluate the red-black tree (non-CF tree) and the Pugh skip list (non-CF skip list) to compare our CF tree and CF skip list against on 2^{12} elements with 5% effective updates. Figure 6.6(a) and Figure 6.6(b) depict the speedup of using the CF skip list over using the non-CF skip list (resp. CF tree over using the non-CF tree) for each of the considered transactional memory implementations. Using transaction-based CF data structures as opposed to default ones clearly speeds up the performance of all transactional memories. The benefit of turning to CF is more important when using trees, which confirms our previous results obtained with our lock-based implementations. In particular, the average speedup for all transactional memories and data structures is of 50%. Interestingly, the speedup of using transaction-based CF does not scale much with the number of threads, probably because the overhead induced by transactional memory and Deuce is too heavy for the contention rise to be visible.

When using transactional memory the benefit of these contention friendly algorithms is apparent just by the fact that abstract operation transactions will have smaller read and write sets causing less contention on the data structure and making the operations less likely to abort. Also structural modifications are each broken into a single transaction causing less contention then they would be if they were include in a single large transaction.

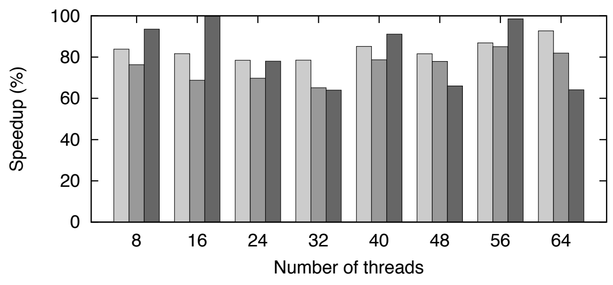
The abstract operations for the tree are very simple, traversals are done in the same way they would be in a sequential algorithm except transactional reads are used. Each physical removal and rotation is performed as a single transaction by the structural adaptation.

In the lock based version of the skip list locks are only used when traversing the bottom level of the structure. Each of the index levels are accessed and modified using only regular read/write operations. This can be applied to the transaction version of the skip list as well. Abstract operation traversals as well as structural modifications to the index level are done outside of transactions. Once an abstract operation traversal has reached the bottom list level a transaction is started, if it arrived on a physically removed node then the operation travels backwards in the list until is reaches a node still in the structure at which point the traversal continues as it would in a sequential list algorithm. Physical removals are done as single transactions by the structural adaptation.

The transactional hash table is very similar to the to the lock-free contention friendly version. Each abstract operation is contained in a single transaction where the compare&swap operations from the lock-free version are replaced by reads and writes. During the rehash operation each bucket rehash is done as a single transaction.



(a) CF-SkipList vs. Pugh Skip List



(b) CF-Tree vs. RB-Tree

Figure 6.6: STM Speedup when using CF data structures instead of their existing counterparts on the Niagara 2 machine

6.10 Pseudo-code and Description

All three data structures share the general structural adaptation code shown in Algorithm 3.

In the normal case the structural adaptation thread works by performing the *background-struct-adaptation* operation constantly traversing the data structure by calling the *restructuring* procedure. Each iteration of this procedure traverses the entire data structure where at each node it might perform some sort of restructuring or a removal. For some data structures after a complete traversal of the structure is done, some restructuring of the entire structure might be needed, this includes the rehash operation of a hash table or the changing of levels of nodes in the skip list.

For backwards compatibility the structural adaptation can also be distributed among the program threads by calling the *distributed-struct-adaptation* operation. In this case each *insert*/*delete* operation will toss a coin, if the value of this coin is greater then some threshold value the thread will then acquire a global structural adaptation lock, and call the *restructuring* procedure before finally releasing the lock, and continuing with its abstract operation.

6.10.1 Tree and Skip List

Skip List As with existing skip list algorithms the structure is made up of many levels of linked lists.

The bottom level of is made up of a doubly linked list of nodes. Each node has a *prev* and *next* pointer, as well as pointer to its key k, an integer *level* indicating the number of levels of linked lists this node has, the *rem* and *del* flags, and a lock.

The upper levels are made up of singly linked lists of IndexItems. Each of these items has a *next* pointer, pointing to the next item in the linked list. A *down* pointer, pointing to the linked list of IndexItems one level below (the bottom level of IndexItems have $\perp$ for their *down* pointers). And a *node* pointer that points to the corresponding node in the Speculation Friendly Skip List.

A per structure array of pointers called *first* is also kept that points to the first element of each level of the skip list. The pointer *top* points to the first element of the highest index of the list, all traversals start from this pointer.

Tree The tree is made up of nodes with each node having left and right child pointers l and r, as well as a pointer to its key k, integers indicating the estimated local height of this node and its children *left-h*, *right-h*, and *local-h*, the *rem* and *del* flags, and a lock.

In addition there is a single pointer *root* that points to the root node of the tree.

6.10.1.1 Skip List Structural Adaptation

The code for the skip list structural adaptation operations is found in Algorithm 1.

The *restructure-node* procedure takes care of removing marked deleted nodes. For each node it checks if it has both a level of 0 and *del* set to `true` then tries to remove the node by calling the *remove-node* procedure. This procedure locks both the node to be removed and the node previous to it in the list in order to not conflict with concurrent *insert* and *delete* operations. The node is then simply removed by changing the previous node's pointer to skip the node. Finally the *rem* flag is set to true and the locks are released.

The *restructure-structure* procedure raises and lowers the levels of the nodes in order to keep the logarithmic traversal cost of the abstract operations. This is done by calling the

Algorithm 1 Skip List Specific Maintenance Operations

```
restructure-node$(node)_s$:
    if $node.level = 0 \cap node.del$ then
        remove$(node.prev, node)$

restructure-structure$()_s$:
    $size \leftarrow$ raise-node-level$()$
    $i \leftarrow 1$
    $count \leftarrow$ raise-index-level$(i)$
    while $count > 2$ do
        $i \leftarrow i+1$
        $count \leftarrow$ raise-index-level$(i)$
    $top \leftarrow first[i]$
    if $\log(size) < i$ then
        lower-index-level$()$
        // Adjust first array index

lower-index-level$()_s$:
    $index \leftarrow first[2].next$
    while $index \neq \perp$ do
        $index.down \leftarrow \perp$
        $index \leftarrow index.next$

remove$(node, next)_s$:
    if $next.level \neq 0$ then
        return false
    lock$(node)$
    if $node.rem \cup \neg node.del$ then
        unlock$(node)$
        return false
    if $node.next \neq next$ then
        unlock$(node)$
        return false
    lock$(next)$
    $next.next.prev \leftarrow node$
    $node.next \leftarrow next.next$
    $next.rem \leftarrow$ true
    unlock$(node)$
    unlock$(next)$
    return true

raise-index-level$(i)_s$:
    $count \leftarrow 0$
    $prev\text{-}tall \leftarrow first[i+1]$
    $index \leftarrow first[i].next$
    while true do
        $next \leftarrow index.next$
        if $next = \perp$ then
            return $count$
        $prev \leftarrow index.prev$
        if $prev.node.level \leq i$
            $\cap index.node.level \leq i$
                $\cap next.node.level \leq i$ then
            // Allocate a new IndexItem
            // called new
            // Set new as the top IndexItem
            // of index.node
            $new.next \leftarrow prev\text{-}tall.next$
            $prev\text{-}tall.next \leftarrow new$
            $index.node.level \leftarrow i+1$
            $prev\text{-}tall \leftarrow new$
        $count \leftarrow count + 1$
        $index \leftarrow index.next$
```

raise-node-level procedure on the bottom level of the skip list and the *raise-index-level* on higher levels. The code for the procedures is practically the same, just *raise-node-level* is performed on nodes while *raise-index-level* is performed on index levels as such only the *raise-index-level* pseudo code is displayed here. The procedures work by simply traversing the entire level i that they are called on if they encounter 3 or more nodes all with height i then the middle of these nodes is raised to height $i+1$. This is performed on each index level starting from the bottom until there are less than 2 nodes on a level.

Due to nodes being removed from the skip list it might be necessary to decrease the number of index levels in the structure. If the *log* of the number of nodes in the structure is less then the height of the structure then the bottom index level is removed. This is done by the *lower-index-level* procedure which simply traverses the second from bottom index level and sets each index item's *down* pointer to $\perp$. Finally the index of the *first* array must be updated to take account the removal of the bottom index level.

6.10.1.2 Tree Structural Adaptation

The code for these operations is found in Algorithm 2.

The *restructure-node* procedure takes care of removing marked deleted nodes as well as performing rotations and propagating balance information upwards in the tree.

Like in the skip list only certain nodes are removed. These are the nodes that have 1 or 0 children and are a majority of the nodes in the tree. This avoids expensive removal operations that require finding and moving a successor node.

In order to do a removal first the parent and the node to be removed are locked(in order to prevent conflicts with concurrent *insert* and *delete* operations) and the *del* flag of the node is

Algorithm 2 Tree Specific Maintenance Operations

```
restructure-node($node$)$_s$:
  if $node.l.del$ then
    remove($node$, false)
  if $node.r.del$ then
    remove($node$, true)
  propagate($node$)
  if $|node.left\text{-}h - node.right\text{-}h| > 1$ then
    // Perform appropriate rotations

restructure-structure()$_s$:
  // Do nothing

propagate($node$)$_s$:
  if $node.l \neq \bot$ then
    $node.left\text{-}h \leftarrow node.l.localh$
  else
    $node.left\text{-}h \leftarrow 0$
  if $node.r \neq \bot$ then
    $node.right\text{-}h \leftarrow node.r.localh$
  else
    $node.right\text{-}h \leftarrow 0$
  $node.localh \leftarrow$
    max($node.left\text{-}h, node.right\text{-}h$) $+ 1$

remove($parent, left\text{-}child$)$_p$:
  if read($parent.rem$) then
    return false
  if $left\text{-}child$ then
    $n \leftarrow$ read($parent.\ell$)
  else
    $n \leftarrow$ read($parent.r$)
  if $n = \bot$ then
    return false
  lock($parent$)
  lock($n$)
  if $\neg$read($n.deleted$) then
    // release locks
    return false
  if ($child \leftarrow$ read($n.\ell$)) $\neq \bot$ then
    if read($n.r$) $\neq \bot$ then
      // Release locks
      return false
  else
    $child \leftarrow$ read($n.r$)
  if $left\text{-}child$ then
    write($parent.\ell, child$)
  else
    write($parent.r, child$)
  write($n.\ell, parent$)
  write($n.r, parent$)
  write($n.rem$, true)
  // release locks
  update-node-heights()
  return true

right-rotate($parent, left\text{-}child$)$_p$:
  if read($parent.rem$) then
    return false
  if $left\text{-}child$ then
    $n \leftarrow$ read($parent.\ell$)
  else
    $n \leftarrow$ read($parent.r$)
  if $n = \bot$ then
    return false
  $\ell \leftarrow$ read($n.\ell$)
  if $\ell = \bot$ then
    return false
  lock($parent$)
  lock($n$)
  lock($\ell$)
  $\ell r \leftarrow$ read($l.r$)
  $r \leftarrow$ read($n.r$)
  // allocate a node called $new$
  $new.k \leftarrow n.k$
  $new.\ell \leftarrow \ell r$
  $new.r \leftarrow r$
  write($\ell.r, new$)
  write($n.rem$, true)
  if $left\text{-}child$ then
    write($parent.\ell, \ell$)
  else
    write($parent.r, \ell$)
  // release locks
  update-node-heights()
  return true
```

checked. The node to be removed has its *left* and *right* child pointers changed so that they point to the parent. This is done to ensure a concurrent operation preempted on this node can still proceed. Next the appropriate parent's child pointer is changed to point to the non-null child of the node to be removed (if any). Finally the *rem* flag is set to *true*.

The structural adaptation is also responsible for keeping the tree well balanced. This is done by doing local rotations. Deciding to do a rotation is based on a local estimated height values. The height values are propagated from the leaves to the root by the *propagate* procedure. This procedure is executed per node and simply reads the *l-height* values of its left and right children, before updating its local values and setting its local *l-height* value to 1 greater then the maximum height of its children. If the absolute value of a nodes left and right heights is at least two then an appropriate rotation is performed. Double rotations are performed as two separate single rotations.

In a traditional rotation operation one node is always rotated downwards. If a concurrent traversal operation is preempted on this node then either it might have to abort or rollback in order to ensure it performs a valid traversal or nodes must be locked/marked during traversal.

In order to avoid using locks and aborts/rollbacks, rotations are preformed differently then traditional rotations. A diagram of the new rotation operation is shown in figure 7.2(c). Before performing the rotation the parent node and the node n that will be rotated are locked in order to prevent conflicts with concurrent *insert* and *delete* operations. Instead of actually modifying n, a new node *new* is created that takes *n*'s place in the structure, this node is set have the same values and pointers as *n* would if a rotation was performed as in existing tree data structures. After the rotation, the node *n* has its *rem* flag set (to true in the case of a right rotation and by-left-rot in the case of a left rotation) and the locks are released.

The reason for not modifying *n* is so that concurrent traversals are not set off track. If the node *n* is removed by a right (resp. left) rotation then its left (resp. right) child has a path to all the nodes as it did before the rotation so a traversal preempted on this node can still traverse the tree safely.

Algorithm 3 States and Restructuring of the Generic CF Algorithm

```
1:  State of process $p$:
2:    $structure$, shared pointer to the data
3:        structure
4:    frequency, the frequency of a structural
5:        adaptation

6:  background-struct-adaptation()$_p$:
7:    while true do
8:        // continuous background restructuring
9:        restructuring()

10: distributed-struct-adaptation()$_p$:
11:   toss($coin$)
12:   if $coin$ > frequency then
13:       // restructure now
14:       restructuring()
15:   // ...or restructure later

16: restructuring()$_p$:
17:   $next \leftarrow$ first-in-trav($structure$)
18:   while $next \neq \perp$ do
19:       restructure_node($next$)
20:       $next \leftarrow$ next-in-trav($next$)
21:   restructure-structure()
```

6.10.2 Abstract Operations

The tree and skip list share code for the *contains*, *insert*, *delete* operations displayed in Algorithm 4. These operations might call one of more of the *get_first*, *get_next*, *validate*, *add* procedures which each have specific code for the given data structure, show in Algorithm 6 for the tree and Algorithm 5 for the skip list. None of these additional procedures use locks or other synchronization methods.

Each of the three abstract operations start by calling the *get_first* procedure. This operation returns the root of the tree or the first node of the top index level of the skip list. The operations then traverse the structure using the *get_next* procedure. This procedure either returns the next node in the traversal or $\perp$ if the traversal is done.

The *get_next* procedure traverses the tree by returning the right child if the node's key is lager then k otherwise the left child is returned. If the nodes key is equal to k and the node is not physically removed then $\perp$ is returned. Since locks are not used during traversal the algorithm has to be aware of concurrent rotations. This means returning the right child in case of being preempted on a node that was removed during a left rotation. If the node was removed during a right rotation then the traversal can continue as normal unless it arrives at a child pointer with value $\perp$, in this case it just returns the other child (which is guaranteed to not be $\perp$).

For the skip list the *get_next* procedure traverses the structure just as it would in a sequential algorithm, with the simple exception that is travels backwards in the list using the *prev* pointer in the case of arriving at a node that has been physically removed. If the nodes key is equal to k and the node is not physically removed or if the traversal is at the bottom level and the next node has key greater then k then $\perp$ is returned.

Algorithm 4 Operations of the Generic CF Algorithm

```
State of node $n$:
  $node$ a record with fields:
    $k \in \mathbb{N}$, the node key
    $del \in \{\mathsf{true}, \mathsf{false}\}$, indicates whether
      logically deleted, initially false
    $rem \in \{\mathsf{true}, \mathsf{false}\}$, indicates whether
      physically deleted, initially false
    $lock$, used to lock the node

contains$(k)_p$:
  $node \leftarrow$ get_first($structure$)
  while true do
    $next \leftarrow$ get_next($node, k$)
    if $next = \perp$ then
      if validate($node, k$) then
        break
    else
      $node \leftarrow next$
  $result \leftarrow$ false
  if $node.k = k$ then
    if $\neg node.del$ then
      $result \leftarrow$ true
  return $result$

insert$(k)_p$:
  $node \leftarrow$ get_first($structure$)
  while true do
    $next \leftarrow$ get_next($node, k$)
    if $next = \perp$ then
      lock($node$)
      if validate($node, k$) then
        break
      unlock($node$)
    else
      $node \leftarrow next$
  $result \leftarrow$ false
  if $node.k = k$ then
    if $node.del$ then
      $node.del \leftarrow$ false
      $result \leftarrow$ true
  else
    add_node($node, k$)
    $result \leftarrow$ true
  unlock($node$)
  return $result$

delete$(k)_p$:
  $node \leftarrow$ get_first($structure$)
  while true do
    $next \leftarrow$ get_next($node, k$)
    if $next = \perp$ then
      lock($node$)
      if validate($node, k$) then
        break
      unlock($node$)
    else
      $node \leftarrow next$
  $result \leftarrow$ false
  if $node.k = k$ then
    if $\neg node.del$ then
      $node.del \leftarrow$ true
      $result \leftarrow$ true
  unlock($node$)
  return $result$
```

Once $\perp$ is returned *insert* and *delete* operations protect the last node in the traversal by locking it (locking is not necessary for the *contains* operations as it does not make modifications). Due to concurrent operations this node may not longer be the end of the traversal, therefore the *validate* procedure is performed on this node ensuring that the traversal has stopped at the correct location. The validation checks to make sure that the node has not been physically removed and that no new node has been inserted directly after this node.

If the validation succeeds then the traversal is finished. Otherwise the lock protecting the node is released and the traversal continues.

Finally some additional code is executed depending on the operation.

In the case of a *contains* operation, the key and/or the deleted flag of the node is checked and a boolean is returned.

In the case of the *insert* operation, first the key of the node is checked, if it is equal to the key being search for then the deleted flag of the node is checked (and possibly modified) and a boolean is returned. Otherwise if the key is not equal to the one being searched for then the *add* operation is performed. The code for the *add* operation simply allocates a new node and attaches it to the data structure by modifying a pointer.

In the case of the *delete* operation the key of the node is checked, if it is equal to the key being search for then the deleted flag of the node is checked (and possibly modified) and a boolean is returned. Otherwise false is returned. It should be noted that when these structures are used as maps the deleted flag can be replaced with a pointer to the value object (from the key/map pair). When this pointer is set to $\perp$ the node is considered as deleted.

Algorithm 5 Skip List Specific Operations

```
Additional fields of IndexItem item:
  IndexItem a record with additional fields:
    next, pointer to the next IndexItem
      in the SkipList
    down, pointer to the IndexItem one
      level below in the SkipList
    node, pointer a node in the list at
      the bottom of the SkipList

Additional fields of node n:
  node a record with additional fields:
    next, pointer to the next node in the list
    prev, pointer to the previous node
      in the list
    level, integer indicating the level of
      the node, initialized to 0


State of structure s:
  top, pointer to the first and highest
    level IndexItem in the SkipList
  first, array of pointers to the first item
    of each level in the SkipList
  bottom-index integer indicating the
    level of the bottom IndexItem

get-first()_s:
  return top

get-next(node, k)_s:
  if node is a list node then
    return get-next-node(node, k)
  else
    return get-next-index(node, k)

get-next-index(node, k)_s:
  next ← node.next
  if next.k > k then
    if node.down ≠ ⊥ then
      return node.down
    return node.node
  else if next.k = k then
    return next.node
  return next

get-next-node(node, k)_s:
  if node.rem then
    while node.rem do
      node ← node.prev
  else
    next ← node.next
    if next = ⊥ ∪ next.k > k then
      return ⊥
    else
      return next

validate(node, k)_s:
  if node.rem then
    return false
  if node.next = ⊥ ∪ node.next.key > k then
    return true
  return false

add(node, k)_s:
  // allocate a node called new
  new.key ← k
  new.prev ← node
  new.next ← node.next
  node.next.prev ← new
  node.next ← new
```

6.10.3 Hash Table

The hash table is made up of two pointers to tables: *table* and *new_table*. The second is used during resize operations. Each process also keeps local variable *l_pointer* that points to a table.

Each table contains an array, with each location in the array containing a list of nodes. Each location in the array is initialized to point to $\perp$. The array also has a single special node associated with it called the *dummy* node which is used during resizing.

6.10.3.1 Abstract Operations

The code for these operations is found in Algorithm 7. Each abstract operation starts by calculating the hash value of the key and then calling the *get-first* procedure. This procedure returns

Algorithm 6 Tree Specific Operations

```
Additional fields of node $n$:
  $node$ a record additional with fields:
    $left\text{-}h, right\text{-}h \in \mathbb{N}$, local height of
      left/right child, initially 0
    $\ell, r \in \mathbb{N}$, left/right child pointers,
      initially $\bot$
    $local\text{-}h \in \mathbb{N}$, expected local height,
      initially 1

State of structure $s$:
  $root$, pointer to root

get-first$()_s$:
  return $root$

get-next$(node, k)_s$:
  $rem \leftarrow node.rem$
  if $node.k > k \cup rem = by\text{-}left\text{-}rot$ then
    $next \leftarrow node.right$
  else
    $next \leftarrow node.left$
  if $next = \bot \cap \neg rem$ then
    if $node.k > k$ then
      return $node.left$
    else
      return $node.right$
  return $next$

validate$(node, k)_s$:
  if $node.rem$ then
    return false
  if $node.next.key > k$ then
    $next \leftarrow node.right$
  else
    $next \leftarrow node.left$
  if $next = \bot$ then
    return true
  return false

add$(node, k)_s$:
  // allocate a node called new
  $new.key \leftarrow k$
  if $node.k > k$ then
    $node.right \leftarrow new$
  else
    $node.left \leftarrow new$
```

the first node in the table located at the bucket given by the hash value or $\bot$ in the case that this bucket is empty. The *get-first* procedure might encounter a dummy node, this means that there is a rehash operation going on that has rehashed this bucket, but not yet finished rehashing the entire table. In this case the local table pointer is updated and the bucket is read again.

Once a value is received from the *get-first* procedure the *contains* simply traverses the list looking for a node with key k.

The *insert* operation also traverses the list looking for a node with key k, if none is found then a new node is allocated and is added to the beginning of the list by performing a compare&swap on the bucket. If the compare&swap fails then the operation restarts.

If the *delete* operation locates a node n with key k in the list then it creates a copy of the list from the first node in the list up to node n but not including n. The bucket is then set to first node of this list by performing a compare&swap. If the compare&swap fails then the operation restarts.

6.10.3.2 Structural Adaptations

The code for these operations is found in Algorithm 8. Local node restructuring is not necessary for the hash table.

The structure restructuring consists of two procedures. The first is the *size* operation that simply traverses the structure counting the number of nodes. If the number of nodes has surpassed some threshold then a resize is necessary and the *grow* procedure is called. This procedure starts be creating a new table larger then the previous one by a power of 2. It then goes through the old table rehashing one bucket at a time. At each bucket in the old table it performs the *grow-level* procedure. This procedure makes a copy of each node in the bucket, rehashing them and placing them in their appropriate buckets in the new hash table. The operation then replaces the list in the old table with its dummy node by performing a compare&swap. If the compare and swap fails then the operation is restarted for this level. Once all levels have been rehashed the *table*

Algorithm 7 HashTable Specific Operations

```
Fields of node $n$:
  node a record with fields:
    $k \in \mathbb{N}$, the node key
    hash, hash value for this node
    next, pointer to next node in list

State of structure $s$:
  table, pointer to array, each location in
    the array contains a list
  table.dummy, pointer to dummy node
  table.mask, binary mask
  new-table, pointer to a table used during
    rehash operations
Process local variables:
  l-table local pointer to table

check-table():
  $t2 \leftarrow$ new-table
  $t1 \leftarrow$ table
  if l-table $= t1$ then
    l-table $\leftarrow t2$
  else
    l-table $\leftarrow t1$

get-first$(hash)_s$:
  l-table $\leftarrow$ table
  node $\leftarrow$ l-table[hash&l-table.mask]
  while node $=$ table.dummy do
    check-table()
    node $\leftarrow$ l-table[hash&l-table.mask]
  return node

contains$(k)_s$:
  node $\leftarrow$ get_first(hash$(k)$)
  while node $\neq \perp$ do
    if $k =$ node.k then
      return true
    node $\leftarrow$ node.next
  return false

insert$(k)_s$:
  hash $\leftarrow$ hash$(k)$
  while true do
    first $\leftarrow$ get_first(hash)
    node $\leftarrow$ first
    index $\leftarrow$ l-table[hash&l-table.mask]
    while node $\neq \perp$ do
      if $k =$ node.k then
        return false
      node $\leftarrow$ node.next
    // allocate a node called new
    new.k $\leftarrow k$
    new.next $\leftarrow$ first
    if C&S(l-table[index], first, new) then
      return true

delete$(k)_s$:
  hash $\leftarrow$ hash$(k)$
  while true do
    first $\leftarrow$ get_first(hash)
    node $\leftarrow$ first
    index $\leftarrow$ l-table[hash&l-table.mask]
    while node $\neq \perp$ do
      if $k =$ node.k then
        prev $\leftarrow$ first
        new-first $\leftarrow$ node.next
        while prev $\neq$ node do
          // make a copy prev
          if prev $=$ first then
            // set new-first to the copy
          prev $\leftarrow$ prev.next
        if prev $\neq$ first then
          prev.next $\leftarrow$ node.next
        if C&S(l-table[index], first, new-first) then
          return true
        else
          // Goto start of outter while loop
      node $\leftarrow$ node.next
    return true
```

pointer is modified so that it points to the new table.

6.11 Correctness

Here we present a sketch of the proof that each data structure algorithm satisfies linearizability.

6.11.1 Skip List

Each of the *contains*, *insert*, and *delete* operations call the *validate* procedure. The *validate* procedure may be called multiple times, but it must return true exactly once. This is used as the linearizability point for the proof sketch. Assume k the key provided as input to the abstract operation.

The result of the *contains*, *insert*, and *delete* operations depends on the existence of a node with key k in the set described by the data structure. At any single point in time there is exactly one valid location in the list where a node with key k can exist (Note that a full proof would

Algorithm 8 HashTable Specific Maintenance Operations

```
1: restructure-node$(node)_s$:
2:   // Do nothing

3: restructure-structure$()_s$:
4:   if size() $>$ threshold then
5:     grow()

6: size$()_s$:
7:   $count \leftarrow 0$
8:   for $i \leftarrow 0$; $i < table.length$; $i$++ do
9:     $next \leftarrow table[i]$
10:    while $next \neq \perp$ do
11:      $count \leftarrow count + 1$
12:      $next \leftarrow next.next$
13:  return $count$

14: grow$()_s$:
15:   $new\text{-}table \leftarrow$ allocate a new table
16:   for $i \leftarrow 0$; $i < table.length$; $i$++ do
17:     grow-level$(i)$
18:   $table \leftarrow new\text{-}table$

19: grow-level$(i)_s$:
20:   while true do
21:     $list1 \leftarrow \perp$
22:     $list2 \leftarrow \perp$
23:     $next \leftarrow table[i]$
24:     $first \leftarrow next$
25:     while $next \neq \perp$ do
26:       // make a copy of next
27:       if hash$(next)$&$new\text{-}table.mask = i$
28:         then
29:         // add copy to list1
30:       else
31:         // add copy to list2
32:       $prev \leftarrow prev.next$
33:     $new\text{-}table[i] \leftarrow list1$
34:     $new\text{-}table[i + table.length] \leftarrow list2$
35:     if C&S$[table[i], first, table.dummy]$ then
36:       return
```

require to show this, for example by induction). This location is the *next* pointer of the node in the list with the largest key smaller then k. For the purpose of this proof sketch we will use a node (call this node *corr*) that is either the node in the list with key k or if no node with key k exists the node whose *next* pointer would point to the node with key k.

Any operation that modifies a node must lock the node before it performs any modification. Given that the *validate* operation is called while a node is locked it only needs to be shown that the when *validate* returns true the node locked is *corr*.

The nodes in the list are sorted by their keys and the *prev* pointer is not modified when a node is removed so any removed node will always have a path to a non-removed node with a smaller or equal key. This means that there will always be a path from a node with key smaller then or equal to k to *corr*. Now since the *get-next* procedure will never traverse past a node that has key larger then k the operation it will always have a path to *corr*. If a node that has been removed is reached during traversal the *prev* pointer is followed, otherwise the *next* pointer is followed during traversal so *corr* will be reached eventually.

Before the *validate* operation returns true it first ensures that the locked node has either key k or the node pointed to by the locked node's *next* pointer has a key larger then k. Second it ensures that the node is not removed ($rem = \text{false}$). Therefore when *validate* returns true, the locked node must be *corr*.

6.11.2 Tree

The tree is a bit more complicated because traversals have to deal with rotations as well as removals. Like in the skip list the abstract operations can call the *validate* procedure multiple times, but it must return true exactly once. This is used as the linearizability point for the proof sketch. Assume k is the key provided as input to the abstract operation.

At any point in time there is exactly one valid location in the tree where a node with key k

can exist. This location is the left or right child pointer of a certain node. This pointer points to the node with key k or to $\perp$ if no node with key k exists. For the purpose of this proof sketch we will use a node (call this node *corr*) that is either the node with this pointer (if no node with key k exists) or the node with key k (is a node with key k does exist).

Before the *validate* operation returns true it first ensures that the locked node either has key k or (if the locked node does not have key k) the child pointer from the locked node where k would exist is $\perp$. Second it ensures that the locked node is not removed (*removed* = false).

Now to complete the sketch it is enough to show that the traversal never passes *corr*. Without rotations or removals this is simple. With removals and rotations the idea is to show that a traversal that is preempted on a node modified by a removal or rotation operation has a path to a node at a higher level in the tree so that the traversal still has a path to *corr*. For removals this is clear due to the fact a removed node has both its child pointers point to its parent during removal. Rotations require a bit more detail. In a traditional left/right rotation one node is rotated downward in the tree while either its left or right child is rotated upwards. For rotations in the contention friendly algorithm the child pointers of the node that would normally be rotated downwards (call this node n) are not modified at all. Instead n is removed from the tree (as the previous parent of n now points to one of n's children) and a new node is created taking n's place. This new node is set up exactly how n would be after a traditional rotation. Now since one of n's children was rotated upwards any concurrent traversal preempted on n will still have a path to all the nodes it did before the rotation.

6.11.3 HashTable

The linearization of the hash table relies on two things, first that the *next* pointer of a node is never modified after it is set during creation, and second any successful modification to a bucket happens by performing a single *compare&swap* on the bucket's pointer.

The linearization point of the *contains* operation is when the *get-first* procedure reads the first element of the bucket that is later returned. Since the next pointer of nodes is never changed then when the operation traverses the list it observes a valid state of the list. The same is true for *insert* and *delete* operations that complete without performing a *compare&swap*.

If a *compare&swap* is required, then the lineraization point is when the *compare&swap* returns successfully. Given that the *compare&swap* operations are only performed at the first element of the bucket, if the operation succeeds then the lineraiztion is valid because the list at the bucket has not changed since *get-first* procedure read it.

6.12 Garbage Collection

Nodes that are physically removed from the data structures must be garbage collected. In the tree and skip list nodes are physically removed only by the structural adaptations. This can happen during rotations of the tree, during the lowering of levels in the skip list, or during the *remove* operation of either structure. Nodes of the hash table are physically removed by the abstract *delete* operation or by the rehash structural adaptation operation. Once a node is physically removed it will no longer be pointed to by the data structure meaning that no future operation will traverse these nodes.

Concurrent traversal operations could be preempted on a removed node so the node cannot be freed immediately. In languages with automatic garbage collection these nodes will be freed as soon as all preempted traversals continue past this node. If automatic garbage collection is not available then some additional mechanisms can be used. One possibility is to provide each thread with a local operation counter and a boolean indicating if the tread is currently performing an abstract operation or not. Then any physically removed node can be safely freed as long as each thread is either not performing an abstract operation or if it has increased its counter since the node was removed. Normally this should be done during the structural adaptation.

6.13 Future Work

6.13.1 Lock-freedom

The tree and skip list algorithms presented in this paper use locks. By using locks they are susceptible to problems such as a thread crashing or being descheduled while holding a lock. In order to avoid these problems, certain concurrent algorithms such as have been designed to be lock-free such as [133]. Lock-free algorithms are generally considered to be complex and difficult to program. This paper focuses on the methodology of designing contention friendly data structures rather then deep descriptions of the algorithms. For this reason the algorithms are described using locks and a brief intuition on how to make the algorithms lock free is given here.

Lock free algorithms often rely on atomic synchronization primitives such as compare&swap in order to preform tasks without using locks. Often a more complex task will require more then just a single atomic operation. In this case one thread might be required to help another thread's operation so that it completes without blocking other operations.

The *insert*, *delete*, *contains* operations of the contention friendly data structures are simple enough to only require at most one compare&swap operation to complete. For an *insert* this might be performing a compare&swap on a pointer and for a *delete* this might be performing a compare&swap on a flag.

The structural adaptation thread takes care of the more complex operations such as removals and modifications to the structure, operations that might require more then just a single compare&swap. Consider a *remove* operation, the structural adaptation thread will initiate the removal by performing a compare&swap to flag the node letting other processes know that it will be removed. The actual removal is then preformed which requires several more compare&swaps. Before the removal is completed another thread might concurrently traverse the flagged node while searching for some key at a different location in the structure. Like in the lock based algorithms this is fine and the traversal can continue as normal. On the other hand a concurrent traversal might need to perform its operation at the location of the node being removed, for example it might need to insert a new node just after the node. In this case the traversal will help the structural adaptation thread with the removal before completing its operation.

In the lock-free version of the concurrent friendly skip list the structural adaptation thread is in charge of removals and raising and lowers of heights of nodes. The raising and lowering of heights can be done just as in the lock based version since no synchronization is required. The removal of nodes uses the same process as in [133] except the structural adaptation thread will start the removal and program threads will help as necessary.

In the tree removals as well as rotations are performed by the structural adaptation thread and might be helped by program threads. Each of these operations is started by the structural adaptation thread performing a compare&swap to flag the node. The only non-blocking implementation of a binary search tree that we know of is presented in [101]. This tree is *leaf-oriented* where keys are stored in leaf nodes.

Even though we do belive it would be possible to implemnt these lock-free algorithms we do expect them to be very complex and would require more investigation.

6.13.2 Structural Adaptation Throttling

In the default version of these algorithms a separate structural adaptation thread is created that continually traverses the tree performing structural modification as necessary. In workloads with low update rates this constant traversal will not have any modification to perform, wasting computation. Even if there are extra unused cores this extra computation may be unwanted due to additional power consumption. Future work should include studying way to throttle the structural adaptation dynamically based on the workload. This could mean putting the thread to sleep during periods of low update rate or even starting and stopping the structural adaptation thread entirely. In largely parallel workloads with high update rates it might even be beneficial to have multiple structural adaptation threads that can be started and stopped at will.

6.13.3 Distributed Structural Adaptation

Each structural adaption is a short local operation, yet each round of structural adaptation is done by a complete traversal of the data structure. Some might argue that if all the hardware of a system is already in use by program threads then why not break the structural adaptation into smaller structural adaptations and distribute them over the abstract modifications. The reason for not doing this is twofold.

Firstly even though each structural adaption is a very local operation, they use global information. For example rotations in a tree need balance information that is propagated from the leaves. Since only nodes with height 1 are removed from the skip list the structural adaptation needs to know about the heights of the other nodes before raising the level of a node in order to ensure that the structure does not become unbalanced. Before resizing the hash table the structural adaptation should know approximately how many nodes are in the table.

Secondly the structural adaptations are in some cases more costly (in terms of computation, not contention) then in existing data structures. For example a rotation requires allocating a new node, choosing the levels of nodes in the skip list requires previously traversing the other nodes to know their height, and resizing the hash table first requires counting the nodes.

Such algorithms with distributed structural adaptation might be possible, but have not been examined here, but could be interesting to study in the future.

Chapter 7

A Contention Friendly Binary Search Tree

7.1 Lock Based Algorithm

7.2 ***Lock Free Algorithm***

7.3 Transactional Memory Based Algorithm

7.4 Introduction

The multicore era is changing the way we write concurrent programs. In such context, concurrent data structures are becoming a bottleneck building block of a wide variety of concurrent applications. Generally, they rely on invariants [91] which prevent them from scaling with multiple cores: a tree must typically remain sufficiently balanced at any point of the concurrent execution.

New programming constructs like transactions [78, 92] promise to exploit the concurrency inherent to multicore architectures. Most transactions build upon *optimistic synchronization*, where a sequence of shared accesses is executed speculatively and might abort. They simplify concurrent programming for two reasons. First, the programmer only needs to delimit regions of sequential code into transactions or to replace critical sections by transactions to obtain a safe concurrent program. Second, the resulting transactional program is reusable by any programmer, hence a programmer composing operations from a transactional library into another transaction is guaranteed to obtain new deadlock-free operations that execute atomically. By contrast, *pessimistic synchronization*, where each access to some location x blocks further accesses to x, is harder to program with [89, 90] and hampers reusability [77, 75].

Yet it is unclear how one can adapt a data structure to access it efficiently through transactions. As a drawback of the simplicity of using transactions, the existing transactional programs spanning from low level libraries to topmost applications directly derive from sequential or pessimistically synchronized programs. The impacts of optimistic synchronization on the execution is simply ignored.

To illustrate the difference between optimistic and pessimistic synchronizations consider the example of Figure 7.1 depicting their step complexity when traversing a tree of height h from its

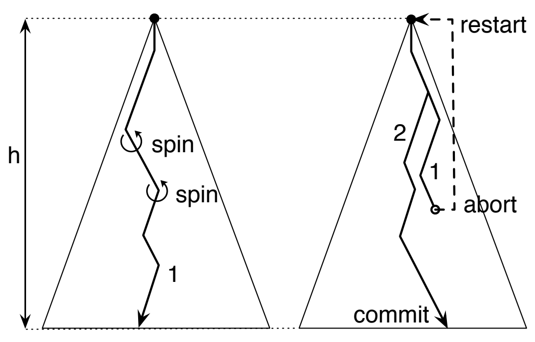


Figure 7.1: A balanced search tree whose complexity, in terms of the amount of accessed elements, is **(left)** proportional to h in a pessimistic execution and **(right)** proportional to the number of restarts in an optimistic execution

root to a leaf node. On the left, steps are executed pessimistically, potentially spinning before being able to acquire a lock, on the path converging towards the leaf node. On the right, steps are executed optimistically and some of them may abort and restart, depending on concurrent thread steps. The pessimistic execution of each thread is guaranteed to execute $O(h)$ steps, yet the optimistic one may need to execute $\Omega(hr)$ steps, where r is the number of restarts. Note that r depends on the probability of conflicts with concurrent transactions that depends, in turn, on the transaction length and h. Although it is clear that a transaction must be aborted before violating the abstraction implemented by this tree, e.g., inserting k successfully in a set where k already exists, it is unclear whether a transaction must be aborted before slightly unbalancing the tree implementation to strictly preserve the balance invariant.

We introduce a *speculation-friendly* tree as a tree that transiently breaks its balance structural invariant without hampering the abstraction consistency in order to speed up transaction-based accesses. Here are our contributions.

- We propose a speculation-friendly binary search tree data structure implementing an associative array and a set abstractions and decoupling the operations that modify the abstraction (we call these *abstract transactions*) from operations that modify the tree structure itself but not the abstraction (we call these *structural transactions*). An abstract transaction either inserts or deletes an element from the abstraction and in certain cases the insertion might also modify the tree structure. Some structural transactions rebalance the tree by executing a distributed rotation mechanism: each of these transactions executes a local rotation involving only a constant number of neighboring nodes. Some other structural transactions unlink and free a node that was logically deleted by a former abstract transaction.

- We prove the correctness (i.e., linearizability) of our tree and we compare its performance against existing transaction-based versions of an AVL tree and a red-black tree, widely used to evaluate transactions [99, 80, 67, 108, 73, 93, 72]. The speculation-friendly tree improves by up to 1.6× the performance of the AVL tree on the micro-benchmark and by up to 3.5× the performance of the built-in red-black tree on a travel reservation application, already well-engineered for transactions. Finally, our speculation-friendly tree performs similarly to a non-rotating tree but remains robust in face of non-uniform workloads.

- We illustrate (i) the portability of our speculation-friendly tree by evaluating it on two different Transactional Memories (TMs), TinySTM [73] and $\mathscr{E}$-STM [131] and with different configuration settings, hence outlining that our performance benefit is independent from the transactional algorithm it uses; and (ii) its reusability by composing straightforwardly the remove and insert into a new move operation. In addition, we compare the benefit of relaxing data structures into speculation-friendly ones against the benefit of only relaxing transactions, by evaluating elastic transactions. It shows that, for a particular data structure, refactoring its algorithm is preferable to refactoring the underlying transaction algorithm.

The paper is organized as follows. In Section 7.5 we describe the problem related to the use of transactions in existing balanced trees. In Section 7.6 we present our speculation-friendly binary search tree. In Section 8.8 we evaluate our tree experimentally and illustrate its portability and reusability. In Section 8.5 we describe the related work and Section 8.9 concludes.

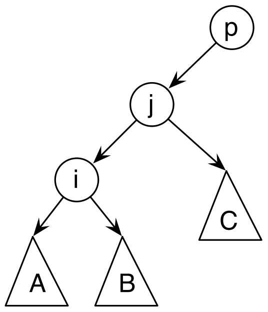
7.5 The Problem with Balanced Trees

In this section, we focus our attention on the structural invariant of existing tree libraries, namely the *balance*, and enlighten the impact of their restructuring, namely the *rebalancing*, on contention.

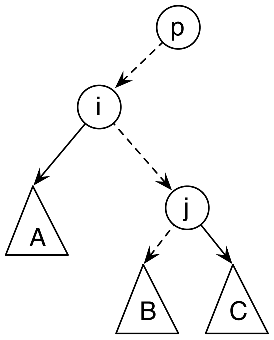
Trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if their difference exceeds a given threshold, the structural invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [61] do not tolerate the longest length to exceed the shortest by 2 whereas red-black trees [64] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations.

Generally, one takes an existing tree algorithm and encapsulates all its accesses within transactions to obtain a concurrent tree whose accesses are guaranteed atomic (i.e., linearizable), however, the obtained concurrent transactions likely *conflict* (i.e., one accesses the same location another is modifying), resulting in the need to abort one of these transactions which leads to a significant waste of efforts. This is in part due to the fact that encapsulating an *update* operation (i.e., an insert or a remove operation) into a transaction boils down to encapsulating four phases in the same transaction:

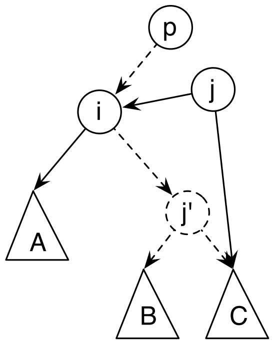
1. the modification of the abstraction,
2. the corresponding structural adaptation,
3. a check to detect whether the threshold is reached and
4. the potential rebalancing.



(a) Initial tree



(b) Result of a classical right rotation



(c) Result of the new right rotation

Figure 7.2: The classical rotation modifies node j in the tree and forces a concurrent traversal at this node to backtrack; the new rotation left j unmodified, adds j' and postpones the physical deletion of j

A transaction-based red-black tree An example is the transaction-based binary search tree developed by Oracle Labs (formerly Sun Microsystems) and other researchers to extensively evaluate transactional memories [99, 80, 67, 108, 73, 93, 72] . This library relies on the classical red-black tree algorithm that bounds the step complexity of pessimistic insert/delete/contains. It has been slightly optimized for transactions by removing sentinel nodes to reduce false-conflicts, and we are aware of two benchmark-suite distributions that integrate it, STAMP [67] and synchrobench[1]http://lpd.epfl.ch/gramoli/php/synchrobench.php.

Each of its update transactions encapsulate all the four phases given above even though phase (1) could be decoupled from phases (3) and (4) if transient violations of the balance invariant were tolerated. Such a decoupling is appealing given that phase (4) is subject to conflicts. In fact, the algorithm balances the tree by executing rotations starting from the position where a node is inserted or deleted and possibly going all the way up to the root. As depicted in Figure 7.2(a) and (b), a rotation consists of replacing the node where the rotation occurs by the child and adding this replaced node to one of its subtrees. A node cannot be accessed concurrently by an abstract

[1]http://lpd.epfl.ch/gramoli/php/synchrobench.php

transaction and a rotation, otherwise the abstract transaction might miss the node it targets while being rotated downward. Similarly, rotations cannot access common nodes as one rotation may unbalance the others.

Moreover, the red-black tree does not allow any abstract transaction to access a node that is concurrently being deleted from the abstraction because phases (1) and (2) are tightly coupled within the same transaction. If this was allowed the abstract transaction might end up on the node that is no longer part of the tree. Fortunately, if the modification operation is a deletion then phase (1) can be decoupled from the structural modification of phase (2) by marking the targeted node as logically deleted in phase (1) effectively removing it from the set abstraction prior to unlinking it physically in phase (2). This improvement is important as it lets a concurrent abstract transaction travel through the node concurrently being logically deleted in phase (1) without conflicting. Making things worse, without decoupling these four phases, having to abort within phase (4) would typically require the three previous phases to restart as well. Finally without decoupling only contains operations are guaranteed not to conflict with each other. With decoupling, insert/delete/contains do not conflict with each other unless they terminate on the same node as described in Section 7.6.

To conclude, for the transactions to preserve the atomicity and invariants of such a tree algorithm, they typically have to keep track of a large *read set* and *write set*, i.e., the sets of accessed memory locations that are protected by a transaction. Possessing large read/write sets increases the probability of conflicts and thus reduces concurrency. This is especially problematic in trees because the distribution of nodes in the read/write set is skewed so that the probability of the node being in the set is much higher for nodes near the root and the root is guaranteed to be in the read set.

Illustration To briefly illustrate the effect of tightly coupling update operations on the step complexity of classical transactional balanced trees we have counted the maximal number of reads necessary to complete typical insert/remove/contains operations. Note that this number includes the reads executed by the transaction each time it aborts in addition to the read set size of the transaction obtained at commit time.

Update	0%	10%	20%	30%	40%	50%
AVL tree	29	415	711	1008	1981	2081
Oracle red-black tree	31	573	965	1108	1484	1545
Speculation-friendly tree	29	75	123	120	144	180

Table 7.1: Maximum number of transactional reads per operation on three 2^{12}-sized balanced search trees as the update ratio increases

We evaluated the aforementioned red-black tree, an AVL tree, and our speculation-friendly tree on a 48-core machine using the same transactional memory (TM) algorithm[2]. The expectation of the tree sizes is fixed to 2^{12} during the experiments by performing an insert and a remove with the same probability. Table 7.1 depicts the maximum number of transactional reads per operation observed among 48 concurrent threads as we increase the update ratio, i.e., the proportion of insert/remove operations over contains operations.

[2]TinySTM-CTL, i.e., with lazy acquirement [73].

For all three trees, the transactional read complexity of an operation increases with the update ratio due to the additional aborted efforts induced by the contention. Althought the red-black and the AVL trees objective is to keep the complexity of pessimistic accesses $O(\log_2 n)$ (proportional to 12 in this case), where n is the tree size, the read complexity of optimistic accesses grows significantly (14$\times$ more at 10% update than at 0%, where there are no aborts) as the contention increases. As described in the sequel, the speculation-friendly tree succeeds in limiting the step complexity raise (2.6$\times$ more at 10% update) of data structure accesses when compared against the transactional versions of state-of-the-art tree algorithms. An optimization further reducing the number of transactional reads between 2 (for 10% updates) and 18 (for 50% updates) is presented in Section 7.6.3.

7.6 The Speculation-Friendly Binary Search Tree

We introduce the speculation-friendly binary search tree by describing its implementation of an associative array abstraction, mapping a key to a value. In short, the tree speeds up the access transactions by decoupling two conflict-prone operations: the node deletion and the tree rotation. Although these two techniques have been used for decades in the context of data management [130, 117], our algorithm novelty lies in applying their combination to reduce transaction aborts. We first depict, in Algorithm 9, the pseudocode that looks like sequential code encapsulated within transactions before presenting, in Algorithm 10, more complex optimizations.

7.6.1 Decoupling the tree rotation

The motivation for rotation decoupling stems from two separate observations: (i) a rotation is tied to the modification that triggers it, hence the process modifying the tree is also responsible for ensuring that its modification does not break the balance invariant and (ii) a rotation affects different parts of the tree, hence an isolated conflict can abort the rotation performed at multiple nodes. In response to these two issues we introduce a dedicated rotator thread to complete the modifying transactions faster and we distribute the rotation in multiple (node-)local transactions. Note that our rotating thread is similar to the collector thread proposed by Dijkstra et al. [130] to garbage collect stale nodes.

This decoupling allows the read set of the insert/delete operations to only contain the path from the root to the node(s) being modified and the write set to only contain the nodes that need to be modified in order to ensure the abstraction modification (i.e., the nodes at the bottom of the search path), thus reducing conflicts significantly. Let us consider a specific example. If rotations are performed within the insert/delete operations then each rotation increases the read and write set sizes. Take an insert operation that triggers a right rotation such as the one depicted in Figures 7.2(a)-7.2(b). Before the rotation the read set for the nodes p,j,i is $\{p.\ell,j.r\}$, where ℓ and r represent the left and right pointers, and the write set is $\emptyset$. Now with the rotation the read set becomes $\{p.\ell,i.r,j.\ell,j.r\}$ and the write set becomes $\{p.\ell,i.r,j.\ell\}$ as denoted in the figure by dashed arrows. Due to $p.\ell$ being modified, any concurrent transaction that traverses any part of this section of the tree (including all nodes i, j, and subtrees A, B, C, D) will have a read/write conflict with this transaction. In the worst case an insert/delete operation triggers rotations all the way up to the root resulting in conflicts with all concurrent transactions.

Algorithm 9 A Portable Speculation-Friendly Binary Search Tree

```
1:  State of node $n$:
2:    $node$ a record with fields:
3:      $k \in \mathbb{N}$, the node key
4:      $v \in \mathbb{N}$, the node value
5:      $\ell, r \in \mathbb{N}$, left/right child pointers, initially $\perp$
6:      $left\text{-}h, right\text{-}h \in \mathbb{N}$, local height of left/right
7:          child, initially 0
8:      $local\text{-}h \in \mathbb{N}$, expected local height, initially 1
9:      $del \in \{\mathsf{true}, \mathsf{false}\}$, indicate whether
10:         logically deleted, initially false

11: State of process $p$:
12:   $root$, shared pointer to root

13: find$(k)_p$:
14:   $next \leftarrow root$
15:   while true do
16:     $curr \leftarrow next$
17:     $val \leftarrow curr.k$
18:     if $val = k$ then break
19:     if $val > k$ then $next \leftarrow$ read$(curr.r)$
20:     else $next \leftarrow$ read$(curr.\ell)$
21:     if $next = \perp$ then break
22:   return $curr$

23: contains$(k)_p$:
24:   transaction {
25:     $result \leftarrow$ true
26:     $curr \leftarrow$ find$(k)$
27:     if $curr.k \neq k$ then $result \leftarrow$ false
28:     else if read$(curr.del)$ then $result \leftarrow$ false
29:   } // current transaction tries to commit
30:   return $result$

31: insert$(k, v)_p$:
32:   transaction {
33:     $result \leftarrow$ true
34:     $curr \leftarrow$ find$(k)$
35:     if $curr.k = k$ then
36:       if read$(curr.del)$ then write$(curr.del, \mathsf{false})$
37:       else $result \leftarrow$ false
38:     else // allocate a new node
39:       $new.k \leftarrow k$
40:       $new.v \leftarrow v$
41:       if $curr.k > k$ then write$(curr.r, new)$
42:       else write$(curr.\ell, new)$
43:   } // current transaction tries to commit
44:   return $result$

45: right_rotate$(parent, left\text{-}child)_p$:
46:   transaction {
47:     if $left\text{-}child$ then $n \leftarrow$ read$(parent.\ell)$
48:     else $n \leftarrow$ read$(parent.r)$
49:     if $n = \perp$ then return false
50:     $\ell \leftarrow$ read$(n.\ell)$
51:     if $\ell = \perp$ then return false
52:     $\ell r \leftarrow$ read$(\ell.r)$
53:     write$(n.\ell, \ell r)$
54:     write$(\ell.r, n)$
55:     if $left\text{-}child$ then write$(parent.\ell, \ell)$
56:     else write$(parent.r, \ell)$
57:     update-balance-values()
58:   } // current transaction tries to commit
59:   return true

60: delete$(k)_p$:
61:   transaction {
62:     $result \leftarrow$ true
63:     $curr \leftarrow$ find$(k)$
64:     if $curr.k \neq k$ then
65:       $result \leftarrow$ false
66:     else
67:       if read$(curr.del)$ then $result \leftarrow$ false
68:       else write$(curr.del, \mathsf{true})$
69:   } // current transaction tries to commit
70:   return $result$

71: remove$(parent, left\text{-}child)_p$:
72:   transaction {
73:     if $left\text{-}child$ then
74:       $n \leftarrow$ read$(parent.\ell)$
75:     else
76:       $n \leftarrow$ read$(parent.r)$
77:     if $n = \perp$ or $\neg$read$(n.del)$ then return false
78:     if $(child \leftarrow$ read$(n.\ell)) \neq \perp$ then
79:       if $(child \leftarrow$ read$(n.r)) \neq \perp$ then return false
80:     if $left\text{-}child$ then
81:       write$(parent.\ell, child)$
82:     else
83:       write$(parent.r, child)$
84:     update-balance-values()
85:   } // current transaction tries to commit
86:   return true
```

Rotation As previously described, rotations are not required to ensure the atomicity of the insert/delete/contains operations so it is not necessary to perform rotations in the same transaction as the insert or delete. Instead we dedicate a separate thread that continuously checks for unbalances and rotates accordingly within its own node-local transactions.

More specifically, neither do the insert/delete operations comprise any rotation, nor do the rotations execute on a large block of nodes. Hence, local rotations that occur near the root can still cause a large amount of conflicts, but rotations performed further down the tree are less subject to conflict. If local rotations are performed in a single transaction block then even the rotations that occur further down the tree will be part of a likely conflicting transaction, so instead each local rotation is performed as a single transaction. Keeping the insert/delete/contains and rotate/remove operations as small as possible allows more operations to execute at the same time without conflicts, increasing concurrency.

Performing local rotations rather than global ones has other benefits. If rotations are performed as blocks then, due to concurrent insert/delete operations, not all of the rotations may still be valid once the transaction commits. Each concurrent insert/delete operation might require a certain set of rotations to balance the tree, but because the operations are happening concurrently the appropriate rotations to balance the tree are constantly changing and since each operation only has a partial view of the tree it might not know what the appropriate rotations are. With local rotations, each time a rotation is performed it uses the most up-to-date local information avoiding repeating rotations at the same location.

The actual code for the rotation is straightforward. Each rotation is performed just as it would be performed in a sequential binary tree (see Figure 7.2(a)-7.2(b)), but within a transaction.

Deciding when to perform a rotation is done based on local balance information omitted from the pseudocode. This technique was introduced in [96] and works as follows. *left-h* (resp. *right-h*) is a node-local variable to keep track of the estimated height of the left (resp. right) subtree. *local-h* (also a node-local variable) is always 1 larger than the maximum value of *left-h* and *right-h*. If the difference between *left-h* and *right-h* is greater than 1 then a rotation is triggered. After the rotation these values are updated as indicated by a dedicated function (line 57). Since these values are local to the node the estimated heights of the subtrees might not always be accurate. The propagate operation (described in the next paragraph) is used to update the estimated heights. Using the propagate operation and local rotations, the tree is guaranteed to be eventually perfectly balanced as in [96, 97].

Propagation The rotating thread executes continuously a depth-first traversal to propagate the balance information. Although it might propagate an outdated height information due to concurrency, the tree gets eventually balanced. The only requirement is that a node knows when it has an empty subtree (i.e., when $node.\ell$ is $\perp$, *node.left-h* must be 0). This requirement is guaranteed since a new node is always added to the tree with *left-h* and *right-h* set to 0 and these values are updated when a node is removed or a rotation takes place. Each propagate operation is performed as a sequence of distributed transactions each acting on a single node. Such a transaction first travels to the left and right child nodes, checking their *local-h* values and using these values to update *left-h*, *right-h*, and *local-h* of the parent node. As no abstract transactions access these three values, they never conflict with propagate operations (unless the transactional memory used is inherently prone to false-sharing).

Limitations Unfortunately, spreading rotations and modifications into distinct transactions still does not allow insert/delete/contains operations that are being performed on separate keys to execute concurrently. Consider a delete operation that deletes a node at the root. In order to remove this node a successor is taken from the bottom of the tree so that it becomes the new root. This now creates a point of contention at the root and where the successor was removed. Every concurrent transaction that accesses the tree will have a read/write conflict with this transaction. Below we discuss how to address this issue.

7.6.2 Decoupling the node deletion

The speculation-friendly binary search tree exploits logical deletion to further reduce the amount of transaction conflicts. This two-phase deletion technique has been previously used for memory management like in [117], for example, to reduce locking in database indexes. Each node has a *deleted* flag, initialized to false when the node is inserted into the tree. First, the delete phase consists of removing the given key k from the abstraction—it logically deletes a node by setting a *deleted* flag to true (line 68). Second, the remove phase physically deletes the node from the tree to prevent it from growing too large. Each of these are performed as a separate transaction and the rotating thread is also responsible for garbage collecting nodes (cf. Section 8.7.4).

The deletion decoupling reduces conflicts by two means. First, it spreads out the two deletion phases in two separate transactions, hence reducing the size of the delete transaction. Second, deleting logically node i simply consists in setting the *deleted* flag to true (line 68), thus avoiding conflicts with concurrent abstract transactions that have traversed i.

Find The find operation is a helper function called implicitly by other functions within a transaction, thus it is never called explicitly by the application programmer. This operation looks for a given key k by parsing the tree similarly to a sequential code. At each node it goes right if the key of the node is larger than k (line 19), otherwise it goes left (line 20). Starting from the root it continues until it either finds a node with k (line 18) or until it reaches a leaf (line 21) returning the node (line 22). Notice that if it reaches a leaf, it has performed a transactional read on the child pointer of this leaf (lines 19–20), ensuring that some other concurrent transaction will not insert a node with key k.

Contains The contains operation first executes the find starting from the root, this returns a node (line 26). If the key of the node returned is equal to the key being searched for, then it performs a transactional read of the *deleted* flag (line 28). If the flag is false the operation returns true, otherwise it returns false. If the key of the returned node is not equal to the key being searched for then a node with the key being searched for is not in the tree and false is returned (lines 27 and 30).

Insertion The insert(k, v) operation uses the find procedure that returns a node (line 34). If a node is found with the same *key* as the one being searched for then the *deleted* flag is checked using a transactional read (line 36). If the flag is false then the tree already contains k and false is returned (lines 37 and 44). If the flag is true then the flag is updated to false (line 36) and true is returned. Otherwise if the *key* of the node returned is not equal to k then a new node is allocated and added as the appropriate child of the node returned by the find operation (lines 38-42). Notice that only in this final case does the operation modify the structure of the tree.

Logical deletion The delete uses also the find procedure in order to locate the node to be deleted (line 63). A transactional read is then performed on the *deleted* flag (line 67). If *deleted* is true then the operation returns false (lines 67 and 70), if *deleted* is false it is set to true (line 68) and the operation returns true. If the find procedure does not return a node with the same *key* as the one being searched for then false is returned (line 65 and 70). Notice that this operation never modifies the tree structure.

Consequently, the insert/delete/contains operations can only conflict with each other in two cases.

1. Two insert/delete/contains operations are being performed concurrently on some key k and a node with key k exists in the tree. Here (if at least one of the operations is an insert or delete) there will be a read/write conflict on the node's *deleted* flag. Note that there will be no conflict with any other concurrent operation that is being done on a different key.

2. An insert that is being performed for some key k where no node with key k exists in the tree. Here the insert operation will add a new node to the tree, and will have a read/write conflict with any operation that had read the pointer when it was $\perp$ (before it was changed to point to the new node).

Physical removal Removing a node that has no children is as simple as unlinking the node from its parent (lines 81–83). Removing a node that has 1 child is done by just unlinking it from its parent, then linking its parent to its child (also lines 81–83). Each of these removal procedures is a very small transaction, only performing a single transactional write. This transaction conflicts only with concurrent transactions that read the link from the parent before it is changed.

Upon removal of a node i with two children, the node in the tree with the immediately larger key than i's must be found at the bottom of the tree. This performs reads on all the way to the leaf and a write at the parent of i, creating a conflict with any operation that has traversed this node. Fortunately, in practice such removals are not necessary. In fact only nodes with no more than one child are removed from the tree (if the node has two children, the remove operation returns without doing anything, cf. line 79). It turns out that removing nodes with no more than one children is enough to keep the tree from growing so large that it affects performance.

The removal operation is performed by the maintenance thread. While it is traversing the tree performing rotation and propogate operations it also checks for logically deleted nodes to be removed.

Limitations The traversal phase of most functions is prone to false-conflicts, as it comprises read operations that do not actually need to return values from the same snapshot. Specifically, by the time a traversal transaction reaches a leaf, the value it read at the root likely no longer matters, thus a conflict with a concurrent root update could simply be ignored. Nevertheless, the standard TM interface forces all transactions to adopt the same strongest semantics prone to false-conflicts [75]. In the next paragraphs we discuss how to extend the basic TM interface to cope with such false-conflicts.

7.6.3 Optional improvements

In previous sections, we have described a speculation-friendly tree that fulfills the standard TM interface [81] for the sake of portability across a large body of research work on TM. Now, we propose to further reduce aborts related to the rotation and the find operation at the cost of an additional lightweight read operation, uread, that breaks this interface. This optimization is thus usable only in TM systems providing additional explicit calls and do not aim at replacing but complementing the previous algorithm to preserve its portability. This optimization complementing Algorithm 9 is depicted in Algorithm 10, it does not affect the existing

Algorithm 10 Optimizations to the Speculation-Friendly Binary Search Tree

```
1:  State of node $n$:
2:    node the same record with an extra field:
3:      $rem \in \{$false, true, true_by_left_rot$\}$
4:        indicate whether physically deleted,
5:        initially false

6:  remove($parent, left\text{-}child$)$_p$:
7:    transaction {
8:      if read($parent.rem$) then return false
9:      if $left\text{-}child$ then
10:       $n \leftarrow$ read($parent.\ell$)
11:     else
12:       $n \leftarrow$ read($parent.r$)
13:     if $n = \perp$ or $\neg$read($n.deleted$) then return false
14:     if ($child \leftarrow$ read($n.\ell$)) $\neq \perp$ then
15:       if read($n.r$) $\neq \perp$ then return false
16:     else
17:       $child \leftarrow$ read($n.r$)
18:     if $left\text{-}child$ then
19:       write($parent.\ell, child$)
20:     else
21:       write($parent.r, child$)
22:     write($n.\ell, parent$)
23:     write($n.r, parent$)
24:     write($n.rem$, true)
25:     update-balance-values()
26:   } // current transaction tries to commit
27:   return true

28: find($k$)$_p$:
29:   $curr \leftarrow root$
30:   $next \leftarrow root$
31:   while true do
32:     while true do
33:       $rem \leftarrow$ false
34:       $parent \leftarrow curr$
35:       $curr \leftarrow next$
36:       $val \leftarrow curr.k$
37:       if $val = k$ then
38:         if $\neg$($rem \leftarrow$ read($curr.rem$)) then break
39:       if $val > k \cup rem =$ true_by_left_rot then
40:         $next \leftarrow$ uread($curr.r$)
41:       else $next \leftarrow$ uread($curr.\ell$)
42:       if $next = \perp$ then
43:         if $\neg$($rem \leftarrow$ read($curr.rem$)) then
44:           if $val > k$ then $next \leftarrow$ read($curr.r$)
45:           else $next \leftarrow$ read($curr.\ell$)
46:           if $next = \perp$ then break
47:         else
48:           if $val \leq k$ then $next \leftarrow$ uread($curr.r$)
49:           else $next \leftarrow$ uread($curr.\ell$)
50:     if $curr.k > parent.k$ then $tmp \leftarrow$ read($parent.r$)
51:     else $tmp \leftarrow$ read($parent.\ell$)
52:     if $curr = tmp$ then
53:       break
54:     else
55:       $next \leftarrow curr$
56:       $curr \leftarrow parent$
57:   return $curr$

58: right_rotate($parent, left\text{-}child$)$_p$:
59:   transaction {
60:     if read($parent.rem$) then
61:       return false
62:     if $left\text{-}child$ then
63:       $n \leftarrow$ read($parent.\ell$)
64:     else
65:       $n \leftarrow$ read($parent.r$)
66:     if $n = \perp$ then
67:       return false
68:     $\ell \leftarrow$ read($n.\ell$)
69:     if $\ell = \perp$ then
70:       return false
71:     $\ell r \leftarrow$ read($\ell.r$)
72:     $r \leftarrow$ read($n.r$)
73:     // allocate a new node
74:     $new.k \leftarrow n.k$
75:     $new.\ell \leftarrow \ell r$
76:     $new.r \leftarrow r$
77:     write($\ell.r, new$)
78:     write($n.rem$, true)
79:     // In the case of a left rotate set
80:     // n.rem to true_by_left_rot
81:     if $left\text{-}child$ then
82:       write($parent.\ell, \ell$)
83:     else
84:       write($parent.r, \ell$)
85:     update-balance-values()
86:   } // current transaction tries to commit
87:   return true
```

contains/insert/delete operations besides speeding up their internal find operation. Here the left rotation is not the exact symmetry of the right rotation code.

This change to the algorithm requires that each node has an additional flag indicating whether or not the node has been physically removed from the tree (a node is physically removed during a successful rotate or remove operation). This removed flag can be set to false, true or true_by_left_rot and is initialized to false. In order not to complicate the pseudo code true_by_left_rot is considered to be equivalent to true, only on line 39 of the find operation is this parameter value specifically checked for.

Lightweight reads The key idea is to avoid validating superfluous read accesses when an operation traverses the tree structure. This idea has been exploited by elastic transactions that use a bounded buffer instead of a read set to validate only immediately preceding reads, thus implementing a form of hand-over-hand locking transaction for search structure [131]. We could have used different extensions to implement these optimizations. DSTM [80] proposes early release to force a transaction stop keeping track of a read set entry. Alternatively, the current distribution of TinySTM [73] comprises unit loads that do not record anything in the read set. While we could have used any of these approaches to increase concurrency we have chosen the unit loads of TinySTM, hence the name uread. This uread returns the most recent value written to memory by a committed transaction by potentially spin-waiting on the location until it stops being concurrently modified.

A first interesting result, is that the read/write set sizes can be kept at a size of $O(k)$ instead of the $O(k \log n)$ obtained with the previous tree algorithm, where k is the number of nested contains/insert/delete operations nested in a transaction. The reasoning behind this is as follow: Upon success, a contains only needs to ensure that the node it found is still in the tree when the transaction commits, and can ignore the state of other nodes it had traversed. Upon failure, it only needs to ensure that the node i it is looking for is not in the tree when the transaction commits, this requires to check whether the pointer from the parent that would point to i is $\perp$ (i.e., this pointer should be in the read set of the transaction and its value is $\perp$). In a similar vein, insert and delete only need to validate the position in the tree where they aimed at inserting or deleting. Therefore, contains/insert/delete only increases the size of the read/write set by a constant instead of a logarithmic amount.

It is worth mentioning that ureads have a further advantage over normal reads other than making conflicts less likely: Classical reads are more expensive to perform than unit reads. This is because in addition to needing to store a list keeping track of the reads done so far, an opaque TM that uses invisible reads needs to perform validation of the read set with a worst case cost of $O(s^2)$, where s is the size of the read set, whereas a TM that uses visible reads performs a modification to shared memory for each read.

Rotation Rotations remain conflict-prone in Algorithm 9 as they incur a conflict when crossing the region of the tree traversed by a contains/insert/delete operation. If ureads are used in the contains/insert/delete operations then rotations will only conflict with these operations if they finish at one of the two nodes that are rotated by rotation operation (for example in Figure 7.2(a) this would be the node i or j). A rotation at the root will only conflict with a contains/insert/delete that finished at (or at the rotated child of) the root, any operations that travel further down the tree will not conflict.

Figure 7.2(c) displays the result of the new rotation that is slightly different than the previous one. Instead of modifying j directly, j is unlinked from its parent (effectively removing it from the tree, lines 82–84) and a new node j' is created (line 73), taking j's place in the tree (lines 82–84). During the rotation j has a removed flag that is set to true (line 78), letting concurrent operations know that j is no longer in the tree but its deallocation is postponed. Now consider a concurrent operation that is traversing the tree and is preempted on j during the rotation. If a normal rotation is performed the concurrent operation will either have to backtrack or the transaction would have to abort (as the node it is searching for might be in the subtree A). Using the new rotation, the preempted operation will still have a path to A.

As previous noted the removed flag can be set to one of three values (false, true or true_by_left_rot). Only when a node is removed during a left rotation is the flag set to true_by_left_rot. This is necessary to ensure that the find operation follows the correct path in the specific case that the operation is preempted on a node that is concurrently removed by a left rotation and this node has the same key k as the one being searched for. In this case the find operation must travel to the right child of the removed node otherwise it might miss the node with key k that has replaced the removed node from the rotation. In all other cases the find operation can follow the child pointer as normal.

Find, contains and delete The interesting point for the find operation is that the search continues until it finds a node with the *removed* flag set to false (line 38 and 43). Once the leaf or a node with the same key as the one being searched for is reached, a transactional read is performed on the *removed* flag to ensure that the node is not removed from the tree (by some other operation) at least until the transaction commits. If *removed* is true then the operation continues traversing the tree, otherwise the correct node has been found. Next, if the node is a leaf, a transactional read must be performed on the appropriate child pointer to ensure this node remains a leaf throughout the transaction (lines 44–45). If this read does not return $\perp$ then the operation continues traversing the tree. Otherwise the operation then leaves the nested while loop (lines 38 and 46), but the find operation does not return yet.

One additional transactional read must be performed to ensure safety. This is the read of the parent's pointer to the node about to be returned (lines 50–51). If this read does not return the same node as found previously, the find operation continues parsing the tree starting from the parent (lines 55–56). Otherwise the process leaves the while loop (line 53) and the node is returned (line 57). By performing a transactional read on the parent's pointer we ensure the STM system performs a validation before continuing.

The advantage of this updated find operation is that ureads are used to traverse the tree, it only uses transactional reads to ensure atomicity when it reaches what is suspected to be the last node it has to traverse. The original algorithm exclusively uses transactional reads to traverse the tree and because of this, modifications to the structure of the tree that occur along the traversed path cause conflicts, which do not occur in the updated algorithm. The contains/insert/delete operations themselves are identical in both algorithms.

Removal The remove operation requires some modification to ensure safety when using ureads during the traversal phase. Normally if a contains/insert/delete operation is preempted on a node that is removed then that operation will have to backtrack or abort the transaction. This can be avoided as follows. When a node is removed, its left and right child pointers are set to point to its previous parent (lines 22–23). This provides a preempted operation with a path back to the tree. The removed node also has its *removed* flag set to true (line 24) letting preempted operations know it is no longer in the tree (the node is left to be freed later by garbage collection).

7.6.4 Garbage collection

As explained previously, there is always a single rotator thread that continuously executes a recursive depth first traversal. It updates the local, left and right heights of each node and performs a rotation or removal if necessary. Nodes that are successfully removed are then added to a garbage collection list. Each application thread maintains a boolean indicating a pending operation and a counter indicating the number of completed operations. Before starting a traversal, the rotator thread sets a pointer to what is currently the end of the garbage collection list and copies all booleans and counters. After a traversal, if for every thread its counter has increased or if its boolean is false then the nodes up to the previously stored end pointer can be safely freed. Experimentally, we found that the size of the list was never larger than a small fraction of the size of the tree but theoretically we expect the total space required to remain linear in the tree size.

7.7 Correctness Proof

There are two parts in the proof. First we have to ensure the structure of the tree is always a valid binary search tree. This is important because in a binary search tree at any time there is exactly one correct location for a key k, the term used for such a tree is *valid binary tree*. A binary tree that does not have the previous property is simply called a *binary tree*. Second we have to show the insert, delete, and contains operations are linearizable.

It is important to remember for this proof that when a transaction commits the transactional reads and writes appear as if they have happened atomically and that unit-read operations only return values from previously committed transactions (or the value written by the current transaction, if the transaction has written to the location being read).

Another important part of this proof is the way the tree is first created. It is created with a root node with key ∞ so that all nodes will always be on its left subtree. This node will always be the root (i.e. it will not be modified by rotate or removal operations), this makes simpler operations and proofs.

Binary trees Each node has two boolean state variables. When the variable *deleted* is false it entails that the value of *node.key* is in the set represented by the tree. When it is true it entails that the value of *node.key* is not in the set represented by the tree. When the variable *removed* is false it entails that the node exists in the tree (meaning a path exists from the root of the tree to the node). When it is true (or true_by_left_rot) it entails that the node does not exist in the tree (meaning no path exists from the root of the tree to the node). For simplicity throughout the pseudo code true_by_left_rot is considered to be equivalent to true, only on line 39 of the find operation is this parameter value specifically checked for.

In the proof we will use the phrase *a node can reach a range of keys* which is explained here. Take the root, from this node there is a path to any node in the tree meaning any key that is in the set is reachable from the root. Furthermore for any key that is not in the tree the root has a path to where it would be inserted (the leaf that would be its parent). This means that the root with key k node has a range $[-\infty,\infty]$. Now its left child has range $[-\infty,k]$ and its right child has range $[k,-\infty]$. Or for example consider some node with range $(10,20]$ and key 14. Its left child will have a range $(10,14)$ and its right child will have a range $(14,20]$. have a path to it.

The phrase *a node n has a path to a range of keys at least as large as some other node n'* is also used in the proof. It means that every key that is in the range of n' is also in the range of n (and possibly some more). For example any node will have a path to a range of keys at least at large as its left child (in fact it has the exact range of its left and right child combined).

Set operations Here traditional operations on the set are defined in the context of transactions. It is important to remember that the TM guarantees a linearization of transactions.

The following definitions are used in defining the operations. Saying a key k is in (not in) the set before the committal of a transaction T means that if some transaction $T1$ performs a contains operation on key k and is serialized as the transaction immediately before T, $T1$ would return true (false).

Saying a key k is in (not in) the set after the committal of a transaction T means that if some transaction $T2$ performs a contains operation on key k and is serialized as the transaction immediately after T, $T2$ would return true (false).

delete For transaction that commits a successful delete operation of key k, before the commit k was in the set and afterwards k is not in the set. For transaction that commits a failed delete operation of key k, before and after the commit k was not in the set.

insert For transaction that commits a successful insert operation of key k, before the commit k was not in the set and after the commit k is in the set. For transaction that commits a failed insert operation of key k, before and after the commit k is in the set.

contains For transaction that commits a successful contains operation of key k, before and after the commit k was in the set. For transaction that commits a failed contains operation of key k, before and after the commit k was not in the set.

Lemma 23 *A node has at most one parent with removed* = false.

Proof. Assume by contradiction that a node has more then one parent with *removed* = false. There are three operation where a node can be given a new parent. First during the insert operations on lines 41–42, but this is a new node so before this line it has no parent. Second during the remove operation on lines 19–21, but by line 8 the other parent has *removed* set to true. Third during the *right-rotate* operation the node l gets a new parent on lines 82–84, but by line 78 the other parent has *removed* set to true. Also during the right_rotate operation the node $n2$ gets l as a parent (line 77), but since it is a new node it has no other parent. (This holds for the left_rotate operation by symmetry) Given this, a node will have at most one parent with *removed* = false. □

Lemma 24 *A node with removed* = false *only has paths from it to other nodes with removed* = false.

Proof. This proof is by induction on the number j of operations done on a node with key k.

The base case is when a new node is created and added to the tree. This can happen in the insert operation. During the operation a new node is created with no children and for itself it has *removed* = false (line 38) and the proof holds.

Now for the induction step from $j = m-1$ to $j = m$, this could be a contains, delete, insert, remove, or rotate operation. First note that the contains and delete operations do not change any children pointers. If this is an insert operation then at $j = m-1$ *left* or *right* must be $\perp$ (line 46 of the find operation) and the proof obviously still holds. If this is a remove operation, then the child of the node is being removed. This means that the new child will either be $\perp$ or the child of the child (lines 14–21), but by induction these nodes have *removed* = false. By symmetry this holds for the right and left children. Otherwise this is a rotate operation. First consider right rotations. By induction we know that node n only has paths to nodes with *removed* = false. After the rotation node l points to nodes that had paths from n before the rotation as well as $n2$ (line 77) which has *removed* = false. $n2$ points to nodes that had paths from n before the rotation (lines 75–76). By symmetry this holds for left rotations. □

Lemma 25 *A node with removed* = false *has at least one parent with removed* = false *that points to it (except the root, as it has no parent).*

Proof. Assume by contradiction that there is no parent with *removed* = false. When a node is first added to the tree it has a parent with *removed* = false (line 43 of the find operation). The only operations that removes links from nodes are the remove (line 24) and rotate (line 78) operations, but in each case when a link is removed, a new link from a different node with *removed* = false is added (lines 19–21 of remove and 82–84 of rotate). □

Lemma 26 *A node with removed* = false *has a path from the root node to it.*

Proof. Given that the root is always the same node and it always has *removed* = false the proof of this lemma follows directly from lemmas 24 and 25. □

Lemma 27 *A node with removed* = true *has no path from the root node to it.*

Proof. By the way the tree is structured the root node always has *removed* = false. Now it follows directly from lemma 24 that there is no path from the root to a node with *removed* = true. □

Lemma 28 *The nodes with removed* = false *make up a single binary tree.*

Proof. From the structure of a node, it can have at most 2 children. Now by lemmas 23, 26, and 27 the proof follows. □

Lemma 29 *A rotation operation on a valid binary search tree results in a valid binary tree of the nodes with removed* = false.

Proof. From Figure 7.2 which describes the *rotation* operation and due to the use of transactional reads/writes we can see that the resulting tree is equivalent to a tree with a classical binary tree rotation performed on it. □

For the following Lemma, we assume that the find operation is performed correctly (this will be proved in a later lemma).

Lemma 30 *Assuming a correct* find *operation, a successful* insert *operation on a valid binary search tree results in a valid binary tree of the nodes with removed* = false.

Proof. There are two cases to consider.

First the key k that we are inserting is already contained in the tree with *removed* = false (lines 37–38 of find). In this case the structure of the tree will not be modified, thus the resulting tree will still be valid.

Second consider that there is no node in the tree with key k. In this case the find operation will return the correct node that will then become the parent of the new node with key k (line 41–42). Using transactional reads, the find operation ensures that the parent node has *removed* = false (line 43) and $\perp$ (line 46) for the child pointer where the new node will be inserted. The new node is then created and added to the tree as the child of the node returned from find. This is done using a transactional write (lines 41–42) which ensures that the value of the pointer was $\perp$ before the write, and finally resulting in a valid tree containing the new node after the transaction commits. □

Lemma 31 *A successful* remove *operation on a valid binary search tree results in a valid binary tree that does not contain the node removed.*

Proof. A removal can only be preformed on a node n with at least one $\perp$ child which is ensured by a transactional read (lines 14–17). Node n being removed is unlinked from its parent (lines 19-21) (the parent is ensured to be in the tree by a transaction read on $removed =$ false) effectively removing it from the tree (lemma 23). If both children of n are set to $\perp$, then the parent's new child becomes $\perp$ (lines 14–21) leaving the tree still valid. If node n has a child c such that $c \neq \perp$, then c becomes the parent's new child (lines 14–21). By lemma 24 this c must have $removed =$ false and by lemma 26 it is part of the valid binary tree during the transaction (until the transaction commits). Thus the resulting tree is still valid. □

Lemma 32 *Modifications to the tree structure are only performed on nodes with* $removed =$ false.

Proof. A rotate, insert or remove operation can modify the tree.

During a right rotate operation the nodes that are modified are n, l and the parent of n (lines 77–84). A transactional read is performed on the remove variable of the parent node (line 61), this along with lemma 24 ensures that $removed =$ false for the parent of n as well as n, and l. By symmetry the left rotate operation also only modifies nodes with $removed =$ false.

During a successful insert operation a new node might be added to the tree, in this case its parent node is modified (lines 41–42), and a transactional read is performed on the parent to ensure that $removed =$ false (line 43 of the find operation).

During a remove operation a node is removed from the tree. A transactional read is performed on the node and its parent (line 8), this along with lemma 24 ensures that $removed =$ false for both nodes. □

Lemma 33 *From a node with* $removed =$ true *there is always a path to some node with* $removed =$ false.

Proof. There are two places where a node can be set to $removed =$ true. First during the remove operation, in this case the node that is removed has both of the nodes child pointers are set to a node with $removed =$ false (lines 22–23). Second during the rotate operation, in this case the node that is removed does not have its child pointers changed. By line 70 n must have at least 1 child and using lemma 24 this child must have $removed =$ false.

Now notice that once the *removed* field of a node is set to true it will never be reverted to false (lemma 32). This along with the use of induction on the length of the path to a node with $removed =$ false completes the proof. □

Lemma 34 *From any node every path leads to a leaf node (or $\perp$).*

Proof. This proof is the same as lemma 33 with a small modification.

First consider a node with $removed =$ false. By lemma 28 it is clear that there is a path from this node to a leaf node.

Now consider a node with $removed =$ true. There are two places where a node's *removed* flag can be set to true. It can either be set in the remove operation, but in this case the node's left and right pointers are set to a node with $removed =$ false (lines 22–23).

Or it can be set in the rotation operation, first notice that in this case the node's left and right pointers are not changed. Then before the rotation the node has *removed* $=$ false and therefore by lemma 24 the (node's left and right) pointers will point to either nodes with *removed* $=$ false or $\perp$.

Now that after a node has had *removed* set to true the node will not be modified again (lemma 32), this along with lemma 28 and the use of induction on the length of the path to a node with *removed* $=$ false or $\perp$ completes the proof. □

The phrase *a node that has removed* $=$ true *from a rotation operation* refers to the node that is no longer in the tree after the rotation. For example in figure 7.2(c) this would be the node j. This phrase is used in the following lemmas.

Lemma 35 *A node n with key k that has removed* $=$ true *(*true_by_left_rot*) from a right_rotation operation for its* ***left*** *child has a path to a range of keys at least as large as n did before the rotation (including the new node $n2$ with key k), and for its* ***right*** *child a path to a range of keys at least as large as the right child before the rotation, or $\perp$.*

Proof. Assume that the node n has key k and before the rotation has a path to a range $[a,b]$. This means that node l (the left child of n) has a path to the range $[a,k]$. Assume node l has key j. The right child of n is either *bot* or some node r with a path to the range $[k,b]$.

After the rotation l keeps it left child with its right child changing to $n2$. $n2$'s right child becomes n's right child, and $n2$'s left child becomes l's old right child. This leaves $n2$ with a path to the range $[j,b]$, and l with a path to the range $[a,b]$.

During the *rotation* operation no modifications are made to node n, except setting *removed* $=$ true. Now n's left child is l giving it a path to the range of keys $[a,b]$. Node n still has the same right child who still has the same range, $[k,b]$. □

Lemma 36 *A node n with key k that has removed* $=$ true *from a $left\_rotation$ operation for its* ***right*** *child has a path to a range of keys at least as large as n did before the rotation (including the new node $n2$ with key k), and for its* ***left*** *child a path to a range of keys at least as large as the left child before the rotation, or $\perp$.*

Proof. This proof follows from symmetry (the left rotation is the mirror of the right rotation) and lemma 35. □

Lemma 37 *A node that has removed* $=$ true *from a* remove *operation has a path to a range of keys as least as large as just before the* remove *operation took place.*

Proof. Assume that the node n (where n is the node to be removed) has key k. Assume that n has a parent node p with range $[a,b]$ with key j. This leaves n with range $[a,k]$ if n is the left child (or $[k,b]$ if n is the right child, the proof of this case will follow by symmetry).

After the removal *removed* will be set to true (line 24 of remove) for n and both its child pointers will point to p (lines 22-23 of remove). Node p will still have a range of $[a,b]$ and will be reachable from n, which completes the proof. □

Lemma 38 *A node that has removed* $=$ true *has either:*

- *for each child a path to a range of keys at least as large as they did when it had removed* $=$ false *or*
- *a single* $\perp$ *child with the other child having a path to a range of keys at least as large as both children before when it had removed* $=$ false*.*

Proof. The proof follows using lemmas 35, 36, and 37 as well as induction on the length of the path from the node to a node with *removed* $=$ false. □

Lemma 39 *A* find *operation on a valid binary search tree always returns the correct location from the valid binary tree.*

Proof. Assume by contradiction that a find operation does not return the correct location from the valid binary tree. By lemma 28 we know that every node with *removed* $=$ false is a node in the valid binary tree so there are two possibilities. The operation never returns or the operation returns the wrong location. First for the operation never returning. From lemma 34 we know that there are no cycles in the path from any node so the operation will not get stuck in an endless loop. In order for the operation to complete it must reach a node with *removed* $=$ false that either matches the key being searched for (lines 37–38) or has $\perp$ as the appropriate child (lines 43–45). Lemma 33 is enough to show that this will always happen and the operation will terminate.

The second possibility where the operation returns the wrong location is more difficult to prove. To this end, we prove by induction on the number of nodes traversed by the operation that the find operation will always have a path to the correct location. In order for the operation to reach the correct location the following must be true: After each traversal from one node to the next the operation must either be at the correct location or have a path from the current location to a range of keys that includes the key being searched for. Once the correct location is reached transactional reads are used to ensure that the location remains correct throughout the transaction (lines 38, and 43–45).

- The base case is easy given that the operation starts from the root which is always part of the valid tree and can reach all nodes with *removed* $=$ false (lemma 26).
- Now the induction step. Assume that after $n-1$ nodes have been traversed the operation has a path to a range of keys that includes the key k that is being searched for. If the $n-1^{th}$ node is the correct location the the proof is done, otherwise the operation must travel to the n^{th} node. Now it must be shown that the after the operation travels to the n^{th} node it is still on the correct path. There are 2 possibilities.
    1. The operation can move from a node with *removed* $=$ false (at the time of the load of the child pointer, lines 40–41). By lemma 24 the child must also have *removed* $=$ false (at the time of the load of the child pointer), in this case the traversal is performed on a valid binary tree (lemma 28). Since the choice of the the child is based on a standard tree traversal the operation must still be on the correct path.
    2. The operation can move from a node with *removed* $=$ true. By the assumption given by induction the $n-1^{th}$ node has a path to the range of keys that includes k. From the $n-1^{th}$ node either the right or left child must be chosen. The node is chosen based on a standard tree traversal with one exception: If the node has the same key k that is being searched for and the node was removed by a left rotation then the right child is

chosen (lines 39–40). With this exception then in all cases if the node is not $\perp$ then by lemma 38 it must have a path to the same range as when it has *removed* = false, which includes k (the $n-1^{th}$ node's range includes k so the n^{th} node's range must also). Otherwise if one child is $\perp$ then the other child is chosen (lines 48–49), which must have a path to a range at least as large as the $n-1^{th}$ node by lemma 38 (which includes k).

□

Theorem 6 *An* insert *operation is valid.*

Proof. The insert operation starts by performing a find operation (line 34). From lemma 39 we know that the find operation will return the correct place. There are two possibilities, either the find operation returns a node in the tree with key k (line 37), or it returns a node in the tree that has $\perp$ for the child where k must be inserted (line 46).

First consider the case where a node with key k is returned. The transactional read on *removed* (line 38 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 36) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns false then the insert operation can return false because the node exists in the set. Otherwise if the read on *deleted* returns true then a transaction performs a transactional write setting *deleted* to false (line 36). The transactional read on *deleted* ensures that k is not in the set before the transaction commits. The transactional write on *deleted* ensures that k is in the set after the transaction commits.

Second consider the case where a node with $key \neq k$ is returned. On lines 44–45 of the find operation a transactional read has been done on the child pointer of this node, returning $\perp$, which must be valid throughout the transaction. The transactional read on *removed* (line 43 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. By lemma 28 this node belongs to a correct binary tree, this along with lemma 39 ensures that the child of this node is the only place where a node with key k could exist (and since the value of the child pointer of $\perp$ a node with key k is not in the tree). A new node with key k is created and then added to the tree using a transactional write to the child pointer (lines 41–42). The *removed* and *deleted* fields of a newly created node can only be false so when the transaction commits the new node will be in the tree and k will be in the set. □

Theorem 7 *A* contains *operation is valid.*

Proof. The contains operation starts by performing a find operation (line 26). From lemma 39 we know that the find operation will return the correct place. There are two possibilities, either the find operation returns a node in the tree with key k (line 37), or it returns a node in the tree that has $\perp$ for the child where k could exist (line 43).

First consider the case where a node with key k is returned. The transactional read on *removed* (line 38 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 28) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns false then the *contains* operation can return true because the node exists in the set otherwise false can be returned because the node does not exist in the set.

Second consider the case where a node with *key* $\neq k$ is returned. On lines 44–45 of the find operation a transactional read has been done on the child pointer of this node, returning $\perp$, and due to the transactional read the pointer must remain as $\perp$ throughout the transaction. The transactional read on *removed* (line 43 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. By lemma 28 this node belongs to a correct binary tree, this along with lemma 39 ensures that the child of this node is the only place where a node with key *k* could exist (and since the value of the child pointer is $\perp$ a node with key *k* is not in the tree). The contains operation can then return false. □

Theorem 8 *A* delete *operation is valid.*

Proof. The delete operation is almost the same as the contains operation. The only difference is in the case where a node with key *k* is returned from the find operation. The transactional read on *removed* (line 38 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 67) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns true then the delete operation can return false because the node does not exist in the set. Otherwise *k* is in the set and a transactional write is performed setting the nodes *deleted* variable to true. Since *removed* is false this node is part of the valid tree so it is the only place where key *k* can be so by setting *deleted* to true *k* must not be in the set. □

7.8 Experimental Evaluation

We experimented our library by integrating it in (i) a micro-benchmark of the synchrobench suite to get a precise understanding of the performance causes and in (ii) the tree-based vacation reservation system of the STAMP suite and whose runs sometimes exceed half an hour. The machine used for our experiments is a four AMD Opteron 12-core Processor 6172 at 2.1 Ghz with 32 GB of RAM running Linux 2.6.32, thus comprising 48 cores in total.

7.8.1 Testbed choices

We evaluate our tree against well-engineered tree algorithms especially dedicated to transactional workloads. The red-black tree is a mature implementation developed and improved by expert programmers from Oracle Labs and others to show good performance of TM in numerous papers [99, 80, 67, 108, 73, 93, 72]. The observed performance is generally scalable when contention is low, most of integer set benchmarks on which they are tested consider the ratio of attempted updates instead of effective updates. To avoid the misleading (attempted) update ratios that capture the number of calls to potentially updating operations, we consider the *effective* update ratios of synchrobench counting only modifications and ignoring the operations that fail (e.g., remove may fail in finding its parameter value thus failing in modifying the data structure).

The AVL tree we evaluate (as well as the aforementioned red-black tree) is part of STAMP [67]. As mentioned before one of the main refactoring of this red-black tree implementation is to avoid the use of sentinel nodes that would produce false-conflicts within transactions. This improvement could be considered a first-step towards obtaining a speculation-friendly binary search tree, however, the modification-restructuring, which remains tightly coupled, prevents scalability to high levels of parallelism.

To evaluate performance we ran the micro-benchmark and the vacation application with 1, 2, 4, 8, 16, 24, 32, 40, 48 application threads. For the micro-benchmark, we averaged the data over three runs of 10 seconds each. For the vacation application, we averaged the data over three runs as well but we used the recommended default settings and some runs exceeded half an hour because of the amount of transactions used. We carefully verified that the variance was sufficiently low for the result to be meaningful.

7.8.2 Biased workloads and the effect of restructuring

In this section, we evaluate the performance of our speculation-friendly tree on an integer set micro-benchmark providing remove, insert, and contains operations, similarly to the benchmark used to evaluate state-of-the-art TM algorithms [73, 131, 69]. We implemented two set libraries that we added to the synchrobench distribution: our non-optimized speculation-friendly tree and a baseline tree that is similar but never rebalances the structure whatever modifications occur. Figure 7.3 depicts the performance obtained from four different binary search trees: the red-black tree (RBtree), our speculation-friendly tree without optimizations (SFtree), the no-restructuring tree (NRtree) and the AVL tree (AVLtree).

The performance is expressed as the number of operations executed per microsecond. The update ratio varies between 5% and 20%. As we obtained similar results with 2^{10}, 2^{12} and 2^{14} elements, we only report the results obtained from an initialized set of 2^{12} elements. The biased workload consists of inserting (resp. deleting) random values skewed towards high (resp. low) numbers in the value range: the values always taken from a range of 2^{14} are skewed with a fixed probability by incrementing (resp. decrementing) with an integer uniformly taken within $[0..9]$.

On both the normal (uniformly distributed) and biased workloads, the speculation-friendly tree scales well up to 32/40 threads. The no-restructuring tree performance drops to a linear shape under the biased workload as expected: as it does not rebalance, the complexity increases with the length of the longest path from the root to a leaf that, in turn, increases with the number of performed updates. In contrast, the speculation-friendly tree can only be unbalanced during a transient period of time which is too short to affect the performance even under biased workloads.

The speculation-friendly tree improves both the red-black tree and the AVL tree performance by up to $1.5\times$ and $1.6\times$, respectively. The speculation-friendly tree is less prone to contention than AVL and red-black trees, which both share similar performance penalties due to contention.

7.8.3 Portability to other TM algorithms

The speculation-friendly tree is an inherently efficient data structure that is portable to any TM systems. It fulfills the TM interface standardized in [81] and thus does not require the presence of explicit escape mechanisms like early release [80] or snap [68] to avoid extra TM bookkeeping (our uread optimization being optional). Nor does it require high-level conflict detection, like open nesting [85, 118, 62] or transactional boosting [108]. Such improvements rely on explicit calls or user-defined abstract locks, and are not supported by existing TM compilers [81] which limits their portability. To make sure that the obtained results are not biased by the underlying TM algorithm, we evaluated the trees on top of $\mathscr{E}$-STM [131], another TM library (on a 2^{16} sized tree where $\mathscr{E}$-STM proved efficient), and on top of a different TM design from the one used so far: with eager acquirement.

The obtained results, depicted in Figure 7.4 look similar to the ones obtained with TinySTM-CTL (Figure 7.3) in that the speculation-friendly tree executes faster than other trees for all TM settings. This suggests that the improvement of speculation-friendly tree is potentially independent from the TM system used. A more detailed comparison of the improvement obtained using elastic transactions on red-black trees against the improvement of replacing the red-black tree by the speculation-friendly tree is depicted in Figure 7.5(a). It shows that the elastic improvements (15% on average) is lower than the speculation-friendly tree one (22% on average, be it optimized or not).

7.8.4 Reusability for specific application needs

We illustrate the reusability of the speculation-friendly tree by composing remove and insert from the existing interface to obtain a new atomic and deadlock-free move operation. Reusability is appealing to simplify concurrent programming by making it modular: a programmer can reuse a library without having to understand its synchronization internals. While reusability of sequential programs is straightforward, concurrent programs can generally be reused only if the programmer understands how each element is protected. For example, reusing a library can lead to deadlocks if shared data are locked in a different order than what is recommended by the library. Additionally, a lock-striping library may not conflict with a concurrent program that locks locations independently even though they protect common locations, thus leading to inconsistencies.

Figure 7.5(b) indicates the performance on workloads comprising 90% of read-only operations (including contains and failed updates) and 10% move/insert/delete effective update operations (among which from 1% to 10% are move operations). The performance decreases as more move operations execute, because a move protects more elements in the data structure than a simple insert or delete operation and during a longer period of time.

7.8.5 The vacation travel reservation application

We experiment our optimized library tree with a travel reservation application from the STAMP suite [67], called vacation. This application is suitable for evaluating concurrent binary search tree as it represents a database with four tables implemented as tree-based directories (cars, rooms, flights, and customers) accessed concurrently by client transactions.

Figure 7.6 depicts the execution time of the STAMP vacation application building on the Oracle red-black tree library (by default), our optimized speculation-friendly tree, and the baseline no-restructuring tree. We added the speedup obtained with each of these tree libraries over the performance of bare sequential code of vacation without synchronization. (A concurrent tree library outperforms the sequential tree when its speedup exceeds 1.) The chosen workloads are the two default configurations ("low contention" and "high contention") taken from the STAMP release, with the default number of transactions, 8× more transactions than by default and 16× more, to increase the duration and the contention of the benchmark without using more threads than cores.

Vacation executes always faster on top of our speculation-friendly tree than on top of its built-in Oracle red-black tree. For example, the speculation-friendly tree improves performance by up to 1.3× with the default number of transactions and to 3.5× with 16× more transactions. The reason of this is twofold: (i) In contrast with the speculation-friendly tree, if an operation on the red-black tree traverses a location that is being deleted then this operation and the deletion conflict. (ii) Even though the Oracle red-black tree tolerates that the longest path from the root to a leaf can be twice as long as the shortest one, it triggers the rotation immediately after this threshold is reached. By contrast, our speculation-friendly tree keeps checking the unbalance to potentially rotate in the background. In particular, we observed on 8 threads in the high contention settings that the red-black tree vacation triggered around $130,000$ rotations whereas the speculation-friendly vacation triggered only $50,000$ rotations.

Finally, we observe that vacation presents similarly good performance on top of the no-restructuring tree library. In rare cases, the speculation-friendly tree outperforms the no-restructuring tree probably because the no-restructuring tree does not physically remove nodes from the tree, thus leading to a larger tree than the abstraction. Overall, their performance is comparable. With $16\times$ the default number of transactions, the contention gets higher and rotations are more costly.

7.9 Related Work

Aside from the optimistic synchronization context, various relaxed balanced trees have been proposed. The idea of decoupling the update and the rebalancing was originally proposed by Guibas and Sedgewick [106] and was applied to AVL trees by Kessels [112], and Nurmi, Soisalon-Soininen and Wood [138], and to red-black trees by Nurmi and Soisalon-Soininen [119]. Manber and Ladner propose a lock-based tree whose rebalancing is the task of separate maintenance threads running with a low priority [114]. Bougé et al. [96] propose to lock a constant number of nodes within local rotations. The combination of local rotations executed by different threads self-stabilizes to a tree where no nodes are marked for removal. The main objective of these techniques is still to keep the tree depth low enough for the lock-based operations to be efficient. Such solutions do not apply to speculative operations due to aborts.

Ballard [94] proposes a relaxed red-black tree insertion well-suited for transactions. When an insertion unbalances the red-black tree it marks the inserted node rather than rebalancing the tree immediately. Another transaction encountering the marked node must rebalance the tree before restarting. The relaxed insertion was shown generally more efficient than the original insertion when run with DSTM [80] on 4 cores. Even though the solution limits the waste of effort per aborting rotation, it increases the number of restarts per rotation. By contrast, our local rotation does not require the rotating transaction to restart, hence benefiting both insertions and removals.

Bronson et al. [97] introduce an efficient object-oriented binary search tree. The algorithm uses underlying time-based TM principles to achieve good performance, however, its operations cannot be encapsulated within transactions. For example, a key optimization of this tree distinguishes whether a modification at some node i grows or shrinks the subtree rooted in i. A conflict involving a growth could be ignored as no descendant are removed and a search preempted at node i will safely resume in the resulting subtree. Such an optimization is not possible using TMs that track conflicts between read/write accesses to the shared memory. This implementation choice results in higher performance by avoiding the TM overhead, but limits reusability due to the lack of bookkeeping. For example, a programmer willing to implement a size operation would need to explicitly clone the data structure to disable the growth optimization. Therefore, the programmer of a concurrent application that builds upon this binary search tree library must be aware of the synchronization internals of this library (including the growth optimization) to reuse it.

Felber, Gramoli and Guerraoui [131] specify the elastic transactional model that ignores false conflicts but guarantees reusability. In the companion technical report, the red-black tree library from Oracle Labs was shown executing efficiently on top of an implementation of the elastic transaction model, $\mathscr{E}$-STM. The implementation idea consists of encapsulating the (i) operations that locate a position in the red-black tree (like insert, contains, delete) into an *elastic* transaction to increase concurrency and (ii) other operations, like size, into a regular transaction. This approach is orthogonal to ours as it aims at improving the performance of the underlying TM, independently from the data structure, by introducing relaxed transactions. Hence, although elastic transactions can cut themselves upon conflict detection, the resulting $\mathscr{E}$-STM, still suffers from congestion and wasted work when applied to non-speculation-friendly data structures. The results presented in Section 7.8.3 confirm that the elastic speedup is even higher when the tree is speculation-friendly.

7.10 Conclusion

Transaction-based data structures are becoming a bottleneck in multicore programming, playing the role of synchronization toolboxes a programmer can rely on to write a concurrent application easily. This work is the first to show that speculative executions require the design of new data structures. The underlying challenge is to decrease the inherent contention by relaxing the invariants of the structure while preserving the invariants of the abstraction.

In contrast with the traditional pessimistic synchronization, the optimistic synchronization allows the programmer to directly observe the impact of contention as part of the step complexity because conflicts potentially lead to subsequent speculative re-executions. We have illustrated, using a binary search tree, how one can exploit this information to design a speculation-friendly data structure. The next challenge is to adapt this technique to a large body of data structures to derive a speculation-friendly library.

Source Code

The code of the speculation-friendly binary search tree is available at **http://lpd.epfl.ch/gramoli/php/synchrobench.php**http://lpd.epfl.ch/gramoli/php/synchrobench.php.

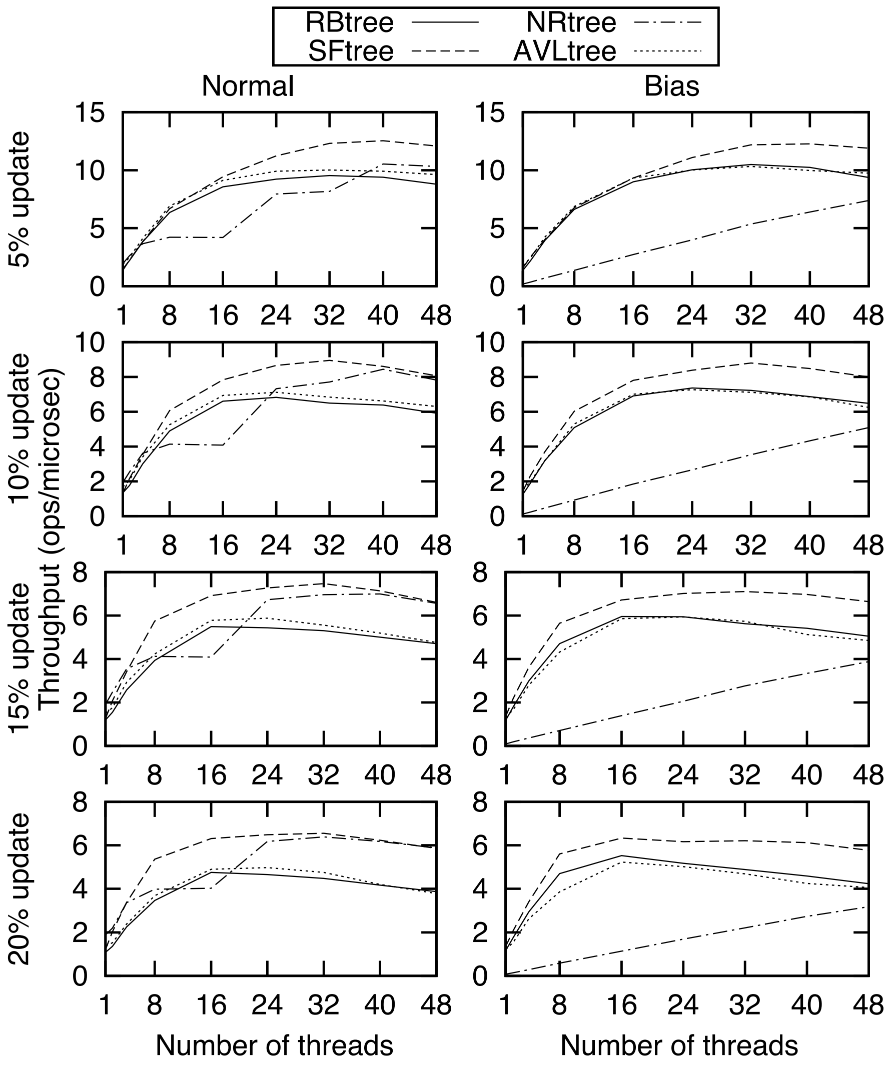


Figure 7.3: Comparing the AVL tree (AVLtree), the red-black tree (RBtree), the no-restructuring tree (NRtree) against the speculation-friendly tree (SFtree) on an integer set micro-benchmark with from 5% **(top)** to 20% updates **(bottom)** under normal **(left)** and biased **(right)** workloads

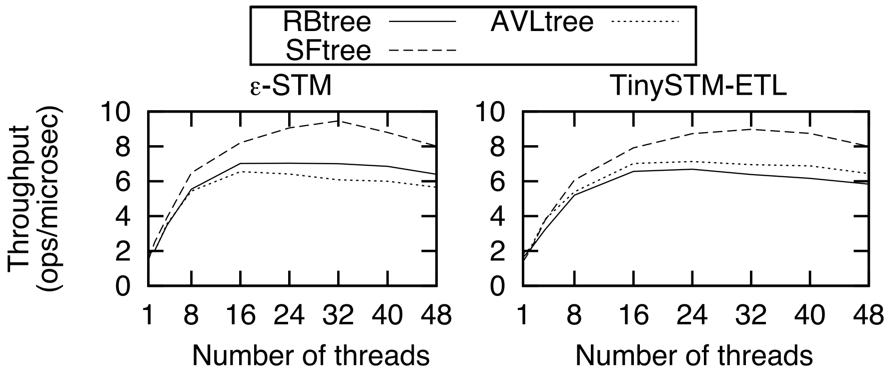
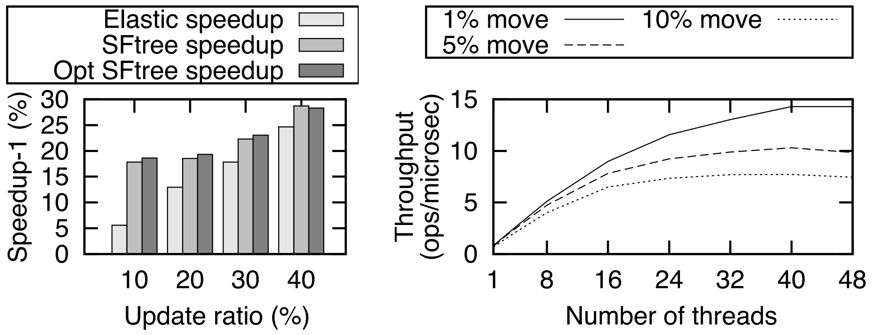


Figure 7.4: The speculation-friendly library running with **(left)** another TM library ($\mathscr{E}$-STM) and with **(right)** the previous TM library in a different configuration (TinySTM-ETL, i.e., with eager acquirement)



(a) Elastic transaction speedup vs. speculation-friendly tree speedup

(b) Performance when reusing the speculation-friendly tree

Figure 7.5: Elastic transaction comparison and reusability

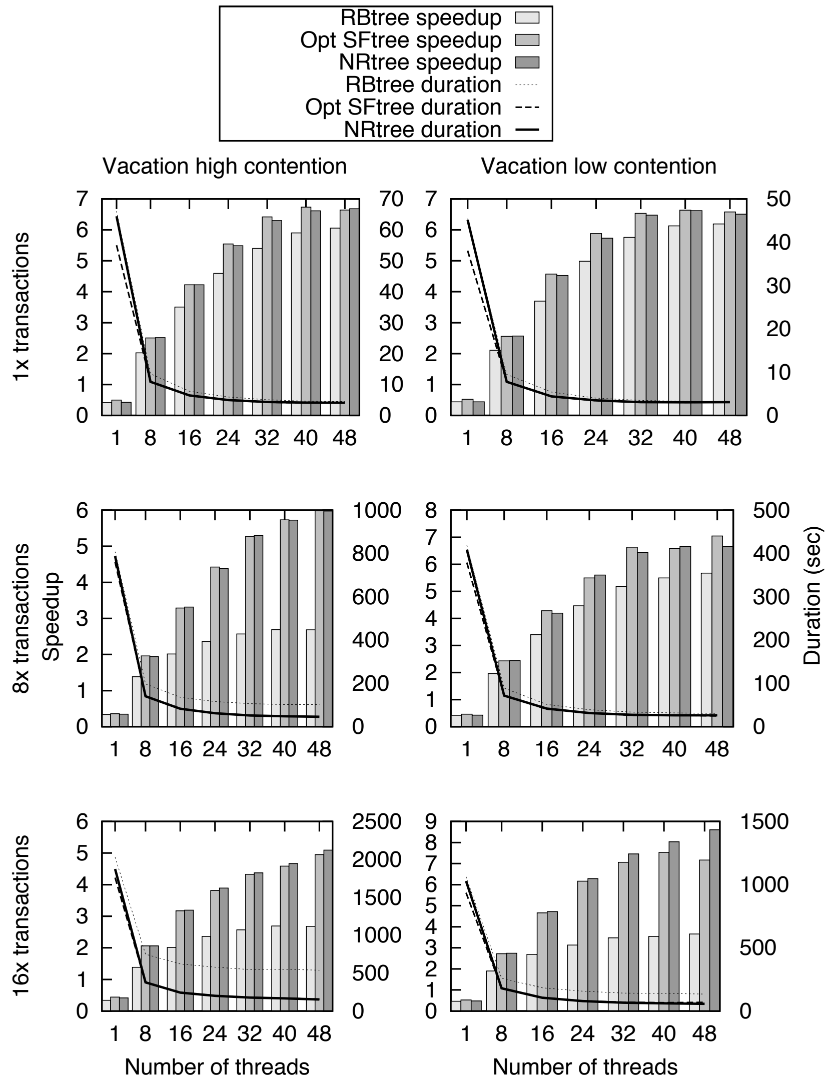


Figure 7.6: The speedup (over single-threaded sequential) and the corresponding duration of the vacation application built upon the red-black tree (RBtree), the optimized speculation-friendly tree (Opt SFtree) and the no-restructuring tree (NRtree) on **(left)** high contention and **(right)** low contention workloads, and with **(top)** the default number of transaction, **(middle)** 8× more transactions and **(bottom)** 16× more transactions

Chapter 8

A Contention Friendly Skip-List

8.1 Lock Based Algorithm

8.2 Lock Free Algorithm

8.3 Transactional Memory Based Algorithm

8.4 Introduction

Multicore architectures are changing the way we write programs. Not only are all computational devices turning multicore thus becoming inherently concurrent, but tomorrow's multicore will embed a larger amount of simplified cores to better handle energy while proposing higher performance, a technology also known as *manycore* [127]. Programmers must thus change their habits to design new concurrent data structures that would be otherwise the bottlenecks of modern every day applications.

The big-oh complexity, which indicates the worst-case amount of converging steps necessary to complete an access, used to prevail in the choice of a particular data structure algorithm running in a sequential context or with limited concurrency. Yet contention has now become an even more important factor of performance drops in today's multicore systems. For example, some concurrent data structures are even so contended that they cannot perform better than bare sequential code, and exploiting additional cores simply make the problem worse [140].

A skip list is a probabilistic data structure to store and retrieve in-memory data in an efficient way, especially used in database systems. In short, a skip list is a linked structure that diminishes the linear big-oh complexity of a linked list with elements having additional shortcuts pointing towards other elements located further in the list [139]. These shortcuts allows operations to complete in $O(\log n)$ steps in expectation. The drawback of employing shortcuts is however to require additional maintenance each time some data is stored or discarded. This causes contention overheads on concurrent skip lists by increasing the probability of multiple *threads* (or processes) interfering on the same shared data. This typically translates into significant performance losses on machine with a large amount of cores.

In the light of the impact of contention on performance, we propose a *Contention-Friendly (CF)* non-blocking skip list that accommodates contention in modern multi-/many-core machines without relaxing the correctness of the abstractions. To this end, we argue for a genuine decoupling of each updating access into an eager abstract modification and a lazy structural adaptation that is selective.

- The *eager abstract modification* consists in modifying the abstraction while minimizing the impact on the skip list itself and returning as soon as possible for the sake of responsiveness.

- The *lazy selective adaptation*, which can be deferred until later, aims at adapting the skip list structure to these changes by re-arranging elements or garbage collecting deleted ones.

More specifically, the aforementioned decoupling translates into splitting an element insertion into the insertion phase at the bottom level of the skip list and the structural adaptation responsible for updating pointers at its higher levels, and an element removal into a logical deletion marking phase and its physical removal and garbage collection.

Additionally, our contention-friendly skip list is *non-blocking*, ensuring that the system as a whole always makes progress.[1] Shortening operations so that they return just after the abstract access diminishes their latency whereas postponing the structural adaptation to avoid temporary load bursts and making it selective to avoid the localized hot-spots helps diminish the contention but also potential starvation.

We prove our algorithm correct and we compare its performance against the Java adaptation by Lea of Harris, Michael and Fraser's algorithms [134, 137, 133]. This implementation is probably one of the mostly used non-blocking skip lists today and is distributed within the Java Development Kit. Our results observed on our 24-core AMD machine shows a 2.5$\times$ speedup.

Section 8.5 describes the related work. Section 8.6 depicts how to make a skip list contention-friendly. Section 8.7 describes in details our contention-friendly non-blocking skip list algorithm. Section 8.8 presents the experimental results and Section 8.9 concludes. Appendix 8.10 depicts additional experimentations and Appendix 8.11 shows our algorithm correct.

8.5 Related Work

Decoupling each data structure modification into multiple tasks has proved beneficial for memory management [130] and efficiency [138, 128], yet this idea was essentially applied to balanced trees but not to diminish contention in skip lists.

Tim Harris proposed to mark elements for deletion using a single bit prior to physically removing them [134]. This bit corresponds typically to the low-order bit of the element reference that would be unused on mostly modern architectures. The decoupling into a logical deletion and a physical removal allowed Harris to propose a non-blocking linked list using exclusively CAS for synchronization. The same technique was used by Maged Michael to derive a non-blocking linked list and a non-blocking hash table with advanced memory management [137] and by Keir Fraser to develop a non-blocking skip list [133].

Doug Lea adapted these algorithms to propose a non-blocking skip list implementation in Java [136]. For the sake of portability, an element is logically deleted by nullifying a value reference instead of incrementing a low-order bit. The resulting algorithm is quite complex and implements a *map*, or dictionary, abstraction. The structure comprises one tower per element

[1]Note that we prefer the term *non-blocking* to the term *lock-free* to denote our targeted progress guarantee. We found it helpful in distinguishing our approach from blocking techniques that do not use locks explicitly.

whose level is determined by a pseudo-random function such that the probability for a tower to have level ℓ is $2^{-O(\ell)}$. A tower of level ℓ comprises $\ell - 1$ *index-items*, one above the other, under which a *node* is used to store the appropriate $\langle key, value \rangle$ pair of the corresponding element. Our implementation uses the same null marker for logical deletion, and we employ the same terminology to describe our algorithm.

Sundell and Tsigas built upon the seminal idea by Valois [143] of constructing non-blocking dictionaries using linked structures. They propose to complement Valois' thesis by specifying a practical non-blocking skip list that implements a dictionary abstraction [141]. The algorithm exploits the logical deletion technique proposed by Harris and uses three standard synchronization primitives that are test-and-set, fetch-and-add and CAS. The performance of their implementation is shown empirically to scale well with the number of threads on an SGI MIPS machine. The logical deletion process that is used here requires that further operations help marking the various levels of a tower upon discovering that the bottommost node is marked for deletion. Further helping operations may be necessary to physically remove the tower.

Fomitchev and Ruppert proposed a non-blocking skip list algorithm whose towers are linked through a doubly linked list [132]. In addition to the original skip list structure [139], it requires backward links to let a traversal potentially backtrack. They also use the logical deletion mechanism and a tower is deleted by first having its bottommost node marked for deletion, then its topmost one. Other operations help removing a tower in an original way by always removing a logically deleted tower to avoid further operations to unnecessarily backtrack. We are unaware of any existing implementation of this algorithm.

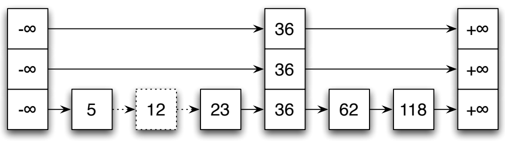
Our non-blocking skip list algorithm uses the same logical deletion technique and the removal process requires some help from another traversal as it is the case in previous skip lists. The main novelty is the decoupling of the abstract modification from the selective structural adaptation to achieve contention-friendliness. In particular, the physical removal applies selectively to towers of low levels to avoid a contention increase at hot spots and some insertions can be accelerated by being done logically. Although it could be distributed, our current structural adaptation is executed by a single thread. This allows us to design a non-blocking skip list in a simpler way than previous approaches. In particular, the adaptations require synchronization only when accessing nodes, at the bottom level.

Finally, transactional memories can be used to implement non-blocking skip lists, however, they may restrict skip list concurrency [133] or block [131]. Note that our notion of contention-friendliness differs from Gadi Taubenfield's contention-sensitivity that was applied to queues [142] as the latter aims at executing an efficient path before switching to a lock-based one when contention raises.

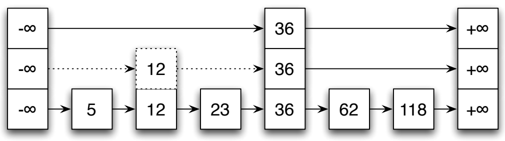
8.6 Towards Contention-Friendliness

In this section, we give an overview of the technique to make the skip list contention-friendly. The crux lies in modifying the traditional skip list structure without relaxing the abstraction or their correctness. Our contention-friendly skip list aims at implementing a correct *map*, or dictionary, abstraction as it represents a common example for storing key-value pairs. The correctness criterion ensured here is linearizability [135].

For the sake of simplicity our map supports only three operations: (i) insert adds a given key-value pair to the map and returns true if the key is not already present; otherwise it returns false; (ii) delete removes a given key and its associated value from the map and returns true if the



(a) Inserting horizontally in the skip list



(b) Adapting vertically the skip list structure

Figure 8.1: Decoupling the eager abstraction insertion from the lazy selective adaptation

key was present, otherwise it returns false; (iii) contains checks whether a given key is present and returns true if so, false otherwise. Note that these operations correspond to the putIfAbsent, remove, and containsKey method of the java.util.concurrent.ConcurrentSkipListMap.

8.6.1 Eager abstract modification

Previous skip lists maintain the node per level distribution so that the probability of a node i to have level ℓ is $\Pr[level_i = \ell] = 2^{-O(\ell)}$, hence each time the abstraction is updated, the invariant is checked and the structure is accordingly adapted as part of the single operation. While an update to the abstraction may only need to modify a single location to become visible, its associated structural adaptation is a global modification that could potentially conflict with any concurrent update.

In order to avoid these additional conflicts, when a node is inserted in the contention-friendly skip list only the bottom level is modified and the additional structural modification is postponed until later. In Appendix 8.11 we show that this abstract modification is sufficient to guarantee linearizability. This decoupling avoids an insertion to update up to $O(\log n)$ levels, this reduces contention and makes the update cost of each operation more predictable.

As an example, assume we aim at inserting an element with key 12 in a skip list. Our insertion consists in updating only the bottom most level of the structure by adding a new node to this level, leading to Figure 8.1(a) where dashed arrows indicate the freshly modified pointers. The key 12 now exists in the set and is visible to future operations, but the process of linking this same node at higher levels is deferred until later.

It is noteworthy that executing multiple abstract modifications without adapting the structure does no longer guarantee the big-oh step complexity of the accesses. Yet this happens only under contention, precisely when the big-oh complexity may not be the predominant factor of performance.

8.6.2 Lazy selective structural adaptation

It is important to guarantee the logarithmic complexity of accesses when there is no contention in the system. Hence when contention stops, the structure needs to be adapted by setting the next pointers at upper levels of the skip list. Figure 8.1(b) depicts the structural adaptation corresponding to the insertion of node 12: the insertion at a higher level of the skip list is executed as a structural adaptation (separated from the insertion), which produces eventually a good distribution of nodes among levels.

Laziness to avoid contention bursts. The structural adaptation is *lazy* because it is decoupled from the abstract modifications and executed by independent threads. Hence many concurrent abstract modifications may have accessed the skip list while no adaptations have completed yet. We say that the decoupling is *postponed* from the system point of view.

This postponement has several advantages whose prominent one is to enable merging of multiple adaptations in one simplified step: only one traversal is sufficient to adapt the structure after a bursts of abstract modifications. Another interesting aspect is that it gives a chance to insertion to execute faster: if the element to be inserted is logically deleted, then the insertion simply needs to logically insert by unmarking it as logically deleted. This avoids the insertion to allocate a new node and to write its value in memory.

Selectivity to avoid contention hot-spots. The abstract modification of a removal simply consists of marking the nodes as deleted without modifying the actual structure. The subsequent structural adaptation selects for removal the nodes whose removal would induce the least contention.

A removal of a node with a high level, say the one with value 36 in Figure 8.1(b), would typically induce more contention than the removal of a node with a lower level, say the one with value 62 spanning a single level. The reason is twofold, first removing a node spanning ℓ levels boils down to updating $O(\ell)$ pointers, hence removing node with value 36 requires to update 3 pointers while node with value 62 requires to update $O(1)$ pointers, second, the organization of the skip list implies that the higher level pointers are traversed more frequently, hence the removal of 36 typically conflicts with every operation concurrently traversing this structure whereas the next pointer of 62 is unlikely to be accessed by a large number concurrent traversals. In the next section, we present an algorithm that removes only towers of height 1.

8.7 The Non-Blocking Skip List

In this section, we present our contention-friendly non-blocking skip list. Section 8.7.1 describes the abstract modifications as well as the contains operation. Section 8.7.2 describes the structural adaptation that is repeatedly executed by a single thread for the sake of simplicity. Section 8.7.3 gives the intuition of the non-blocking guarantee of our algorithm. Section 8.7.4 describes the garbage collection. Section 8.7.5 discusses the distribution on the adaptation on multiple threads. The correctness proof is deferred to the Appendix 8.11.

Figure 11 depicts the algorithm of the eager abstract operations while Figure 12 depicts the algorithm of the lazy selective adaptation. The skip list algorithm is *non-blocking* meaning that there is always at least one thread makes progress after a sufficiently long amount of time. In particular, only CAS operations are used for synchronization. The bottom level of the skip list is made up of a doubly linked list of nodes as opposed to the Java ConcurrentSkipListMap. Each node has a *prev* and *next* pointer, a key k, a value v, an integer *level* indicating the number of levels of linked lists this node has, a *marker* flag indicating whether or not the node is a marker (used during removals).

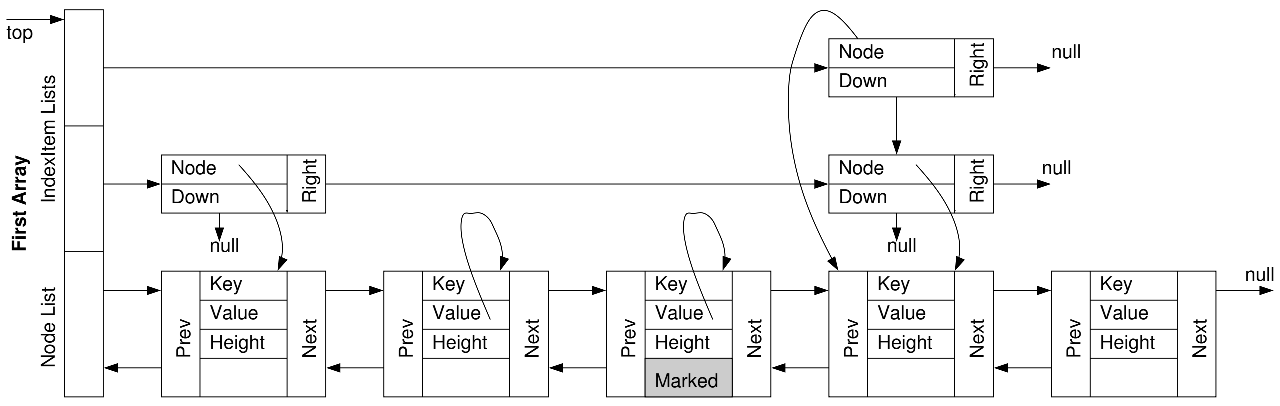


Figure 8.2: Skip list structure

We use the logical deletion technique [134] by nullifying the v field used to hold the value associated with the key of the node. If $v = \perp$, then we say that the node is logically deleted. In order to indicate that a node has been (or is in the process of being) physically removed from the list, the v field is set to point to the node itself (for example a node n has been or is being physically removed if $n.v = n$).

The upper levels are made up of singly linked lists of *index-items*. Each of these items has a *next* pointer, pointing to the next item in the linked list, a *down* pointer, pointing to the linked list of IndexItems one level below (the bottom level of IndexItems have $\perp$ for their *down* pointers), and a *node* pointer that points to the corresponding node at the bottom of the skip list.

A per structure array of pointers called *first* is also kept that points to the first element of each level of the skip list. The pointer *top* points to the first element of the highest index of the list, all traversals start from this pointer. Figure 8.7 shows the structure of the contention friendly skip list where the third node is in the process of being removed and the fourth node is a marker.

8.7.1 Abstract operations

The contains, insert, and delete operations start by traversing the towers using the traverse-tower procedure that traverses the towers similar to a sequential skip list algorithm, moving forward in the list until reaching a node with a larger key than k and then moving down a level. If a node with key k is found then that node is returned immediately, otherwise the operations continues until the bottom of the tower is reached, returning the node of the tower it stops at.

The traversal continues on the bottom list level using the get-next-node procedure. The main differences between this traversal and a sequential algorithm is due to concurrent removals. If a node is encountered in the list that has been marked to be removed (line 89) then the help-remove procedure is called (line 90) or if a node is encountered that has already been removed then the *prev* pointers are used to backtrack into the list.

During the contains operation, if a node with key k is found its value is read (lines 67-69) and either true or false is returned depending on the observed value; if no node with key k is found false is returned.

During a delete operation, if a node with key k is found that is neither removed nor marked deleted (checked on line 21) then a CAS is performed to try marking the node as deleted (line 22).

An interesting implication of separating the structural adaptation is the ability to have lighter insertions. An insert is executed "logically" if it encounters a node with key k that is marked as deleted (lines 34) by unmarking it (lines 35). If no node with key k is found then the insert operation allocates "physically" a new node (line 42) before adding it to the list by performing a CAS operation on the next pointer of the predecessor (line 45). Note that existing skip list algorithms cannot exploit logical insertions as each logical deletion is traditionally followed by a physical removal.

During both insert and delete operations if a CAS operation fails (due to a concurrent modification) then the get-next-node procedure is called again, starting the traversal from the node where the CAS failed.

8.7.2 Structural adaptation

The structural adaptation is executed repeatedly by a dedicated thread, called *adapting thread*. Its first task is to physically remove nodes marked as deleted who have a height of 1. This is done after a successful delete operation, as well as in the adapting thread. The remove operation is more difficult than the abstract operations as it requires three CAS operations. The reason is that a node cannot be safely removed from the list using just one CAS. Consider a node n to be removed that has predecessor and successor nodes *prev* and *next*. If a CAS is performed on *prev.next* removing *node* by changing the pointer's value from *node* to *next* then a concurrent insert operation could have added a new node in between *node* and *next*, leading to a lost update problem [143]. In order to avoid such cases, physical removals are broken into two steps.

First, the v field of the node to be removed is CASed from $\perp$ to point to the node itself on line 100 of the remove operation. This indicates to other threads that the node is going to be removed. Then, the removal is completed in a separate help-remove procedure (which might also be called by a concurrent operation performing a traversal).

We encompass lost insert scenarios by using a special marked node, which is inserted with a CAS just after the node to be removed (lines 112-113), similar to Lea's ConcurrentSkipListMap. In order to distinguish a marked node from other nodes it has its *marked* flag set to true and its v field points to itself. Additionally, a validation checks that neither the predecessor nor the successor node is marked before inserting a new node. (lines 87 and 92 of the get-next-node procedure and on line 44 of the insert operation). To complete the removal a CAS is performed on the predecessor's next pointer (line 117) removing the node and its marker from the list.

A second task of the adapting thread is to modify the upper levels of nodes in order to ensure the $O(\log n)$ expected traversal time. Since neither removals nor insertions are done as they are in traditional skip lists, calculating the height of a node must also be achieved differently. Existing algorithms call a random function to calculate the heights of nodes, here they are done deterministically while considering that the fundamental structure of a skip list is not designed to be perfectly balanced (as it would be too costly) but rather probabilistically balanced. This is done using the procedure raise-index-level, which is called at each tower level from the bottom level going upwards. Each iteration traverses an entire list level, where each time it observes 3 consecutive nodes whose height equals this level (line 133–135), it raises the level of the middle node by 1 (line 141). Such a technique approximates the targeted number of nodes present at each level, balancing the structure.

The final task of the adapting thread is due to the fact that only towers of height 1 are physically removed and is necessary in the case that "too many" tall nodes are marked as deleted. If the number of nodes of levels greater than 1 that are logically deleted passes some threshold then the lower-index-level procedure is called which removes the entire bottom *index-item* level of the skip list by changing the *down* pointers of the level above to $\perp$ (line 122). Doing this avoids modification to the taller nodes in the list and helps ensure there are not too many marked deleted nodes left in the list. There are no frequent re-balancing going on to the tower, tall nodes will stay tall nodes meaning less contention at the frequently traversed locations of the structure.

The adapting thread continually traverses the structure repeating these structural adaptation procedures as necessary while also physically removing appropriate nodes that were not successfully removed by a delete operation.

8.7.3 Non-Blocking

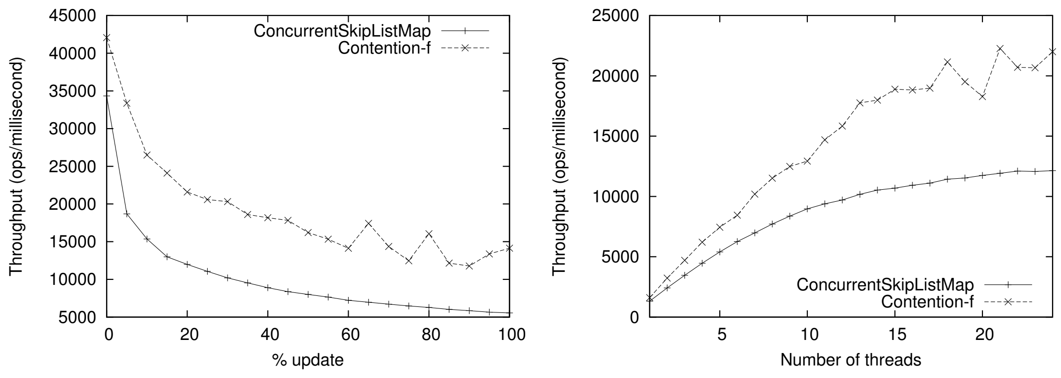
In order for an algorithm to be considered non-blocking at least one thread must make progress after a sufficiently long amount of time. Since none of the operations use locks or any blocking operations the algorithm can be proven to be non-blocking by showing that a thread can only be stuck infinitely in a loop if there is at least one other thread making progress. Many of these loops traverse the skip list following the *next*, *prev* pointers of the nodes or the *down*, *right* pointers of the IndexItems. In Appendix 8.11 we show that the skip list is valid which requires that nodes are sorted in ascending order by their keys. Therefore these traversals can only loop infinitely if there is either an infinite number of nodes being added/removed from the list concurrently.

First we will consider the while loop in the get-next-index procedure. Iterations of this loop traverses forward and down levels of IndexItems.

The help-remove procedure has a while loop on lines 109–114, this loop compares and swaps a marker node into the list, looping until a marker is added successfully. In order for the addition of a marker node to fail an infinite number of times there must be an infinite number of concurrent successful insert operations.

In the get-next-node procedure there is an inner while loop that traverses backward in the list (lines 87–87) and an outer level loop that traverses forward in the list (lines 86–95). An infinite traversal in the inner loop would require an infinite number of node removals which would also require and infinite number of inserts. An infinite number of outer loop iterations would require either a infinitely long list (which would mean an infinite number of inserts) or an infinite number of invocations of the inner while loop. The inner while loop is only invoked when the outer traversal reaches a node in the process of being removed. Therefore in order for the inner loop to be invoked an infinite number of times the outer traversal must encounter one or more marked removed nodes in the list an infinite number of times. Each iteration of the outer loop calls the help-remove procedure, which either successfully physically removes a node from the list or fails due to a concurrent operation succeeding. If it succeeds an infinite number of times then there must be an infinite number of delete operations and if it fails an infinite number of times then there must be an infinite number of concurrent operations modifying the list structure (be it inserts or removals), in either case at least one other thread is making progress.

Both the insert and delete operations contain a while loop. In both operations an iteration of the while loop will always exit unless it calls a CAS that fails. Both operations might perform a CAS on the v field of a node (line 35 of insert and line 22 of delete), but this CAS will only fail due to a concurrent insert or delete successfully performing a CAS on this field. The insert operation might perform a CAS on line 45 changing a node's pointer, this CAS can only fail due to a concurrent successful insert or removal. Therefore an infinite loop in these operations is only possible due to infinite number of successful operations by at least one other thread.



(a) Tolerance to contention of the non-blocking skip lists (b) Scalability of the non-blocking skip lists

Figure 8.3: Comparison of our contention-friendly non-blocking skip list against the JDK concurrent skip list (ConcurrentSkipListMap)

8.7.4 Garbage collection

Nodes that are physically removed from the data structures must be garbage collected. Once a node is physically removed it will no longer be reachable by future operations.

Concurrent traversal operations could be preempted on a removed node so the node cannot be freed immediately. In languages with automatic garbage collection these nodes will be freed as soon as all preempted traversals continue past this node. If automatic garbage collection is not available then some additional mechanisms can be used. One possibility is to provide each thread with a local operation counter and a boolean indicating if the tread is currently performing an abstract operation or not. Then any physically removed node can be safely freed as long as each thread is either not performing an abstract operation or if it has increased its counter since the node was removed. This can be done by the adapting thread. Other garbage collection techniques can be used such as reference counting described in [129].

8.7.5 Distributing the structural adaptation

The algorithm we have presented exploits the multiple computational resources available on today's multicore machines by having a separate adapting thread. It could be adapted to support multiple adapting threads or to make each application thread participate into a distributed structural adaptation (if for example computational resources get limited). Although this approach is appealing to, for example, to maintain the big-oh complexity despite failures, it makes the protocol more complex.

To keep the benefit from the contention-friendliness of the protocol, it is important to maintain the decoupling between the abstract modifications and the structural adaptations. Distributing the tasks of the adapting thread to each application threads should not force them to execute a structural adaptation systematically after each abstract modification. Instead, each application thread could toss a coin after each of its abstract modification to decide whether to run a structural adaptation. This raises an interesting question on the optimal proportion of abstract modifications per adaptation.

The other challenge is to guarantee that concurrent structural adaptations execute safely. This boils down into synchronizing the higher levels of the skip list by using CAS each time a pointer of the high level lists is adapted. An important note is that given the probability distribution of nodes per level in the skip list, the sum of the items in the upper list levels is approximately equal to the number of nodes in the bottom list level. On average the amount of conflicts induced by the skip list with a distributed adaptation could be potentially twice the one of the centralized adaptation. This exact factor depends, however, on the frequency of the distributed structural adaptation.

Finally, to distribute the structural adaptation each thread could no longer rely on the global information regarding the heights of other nodes. To recover to the probability distribution of item to levels without heavy inter-threads synchronization, a solution would be to give up the deterministic level computation adopted in the centralized version and to switch back to the traditional probabilistic technique: each application thread inserting a new node would simply choose a level ℓ with probability $2^{-O(\ell)}$.

8.8 Evaluation

Here we compare our skip list to the java.util.concurrent skip list on a multi-core machine. Additional experiments are deferred to Appendix 8.10. The machine is an AMD with two 12-core processors, comprising 24 hardware threads in total. For each run we averaged the number of executed operations per millisecond over 5 runs of 5 seconds. Thread counts are from 1 to 24 and the five runs execute successively as part the same JVM for the sake of warmup. We used Java SE 1.6.0 12-ea in server mode and HotSpot JVM 11.2-b01.

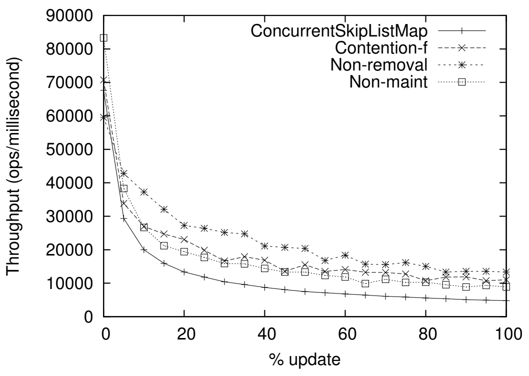
The approximate number of elements in the set abstraction is 5 thousand with operations choosing from a range of 10 thousand keys, implying that each update operation successfully modify the data structure 50% of the time. As insertions and deletions are executed with the same probability the data structure size remains constant in expectation.

The ConcurrentSkipListMap is Doug Lea's Java implementation relying on Harris, Michael and Fraser algorithms [134, 137, 133]. It comes with JDK 1.6 as part of the java.util.concurrent package. We compare this implementation to our contention-friendly skip list as given in Section 8.7—both implementations are non-blocking.

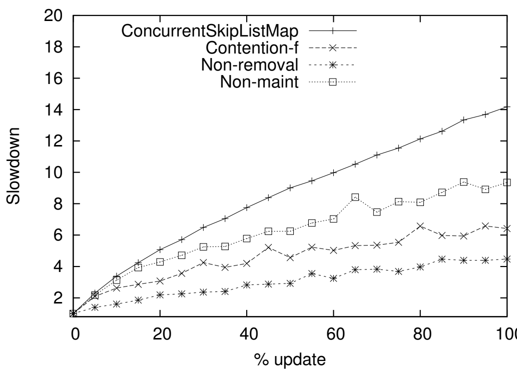
Figure 8.3(a) depicts the tolerance to contention of the algorithms, by increasing the percent of update operations from 0 to 100 (i.e., between 0% and 50% effective structure updates). We can see that the contention-friendly (Contention-f) skip list better tolerates contention than the ConcurrentSkipListMap which results in significantly higher performance.

An interesting result is the gain of using our skip list at 0% update: since the contention-friendly skip list tolerates high contention, it can afford maintaining indices for half of the nodes, so that half of the node have multiple levels. In contrast, ConcurrentSkipListMap maintains the structure so that only one quarter of the nodes have indices, in an attempt to reduce the contention when updates come into play. Actually, our strategy better tolerates contention when updates appear as the contention-friendly skip list is up to 2.5× faster than the ConcurrentSkipListMap.

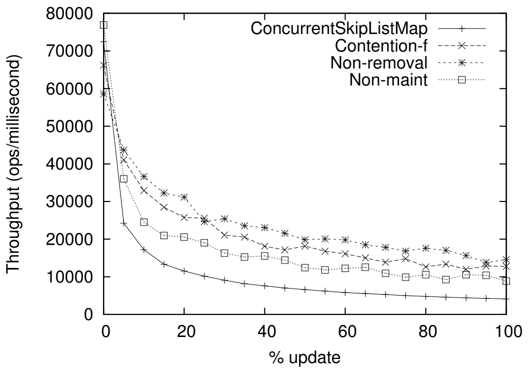
Figure 8.3(b) compares the performance of the skip list algorithms, run with 20% of update operations (i.e., 10% effective structure updates). Although the ConcurrentSkipListMap scales well with the number of threads, the contention-friendly skip list scales better. In fact, the decoupling of the later allows to tolerate the contention raise induced by the growing amount of threads, leading to a performance speedup of up to 1.8×.



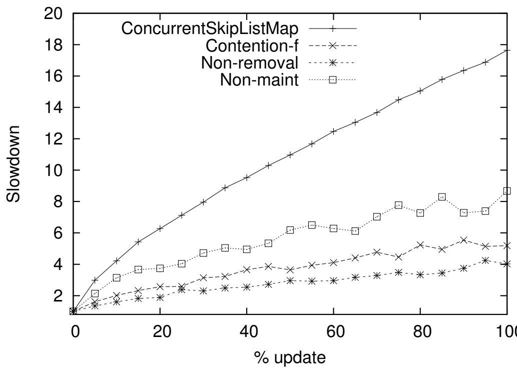
(a) Throughput, range of 128 elements



(b) Slowdown, range of 128 elements



(c) Throughput, range of 128k elements



(d) Slowdown, range of 128k elements

Figure 8.4: Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements

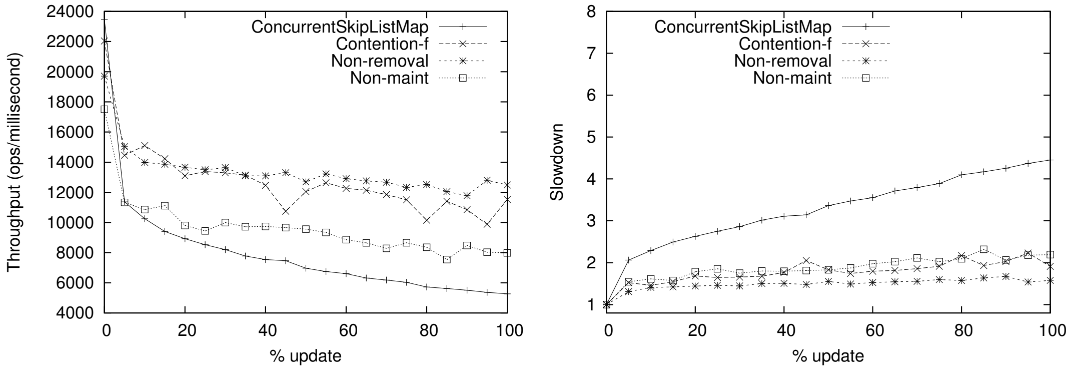
8.9 Conclusion

Multicore programming brings new challenges, like contention, that programmers have to anticipate when developing every day applications. We explore the design of a contention-friendly and non-blocking skip list, keeping in mind that contention is an important cause of performance drop. As future work, we would like to derive new contention-friendly data structures.

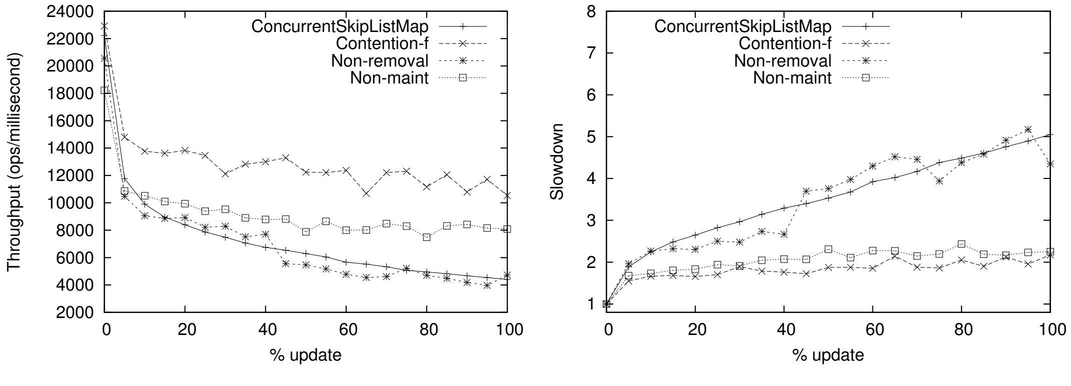
8.10 Additional Evaluation

In this section we present additional performance results. Each benchmark was run 4 times (the average of the 4 runs is presented in the figures) each with a duration of 5 seconds with the JVM being warmed up for 5 seconds prior to running each benchmark. In order to better understand where the benefits of the contention friendly algorithm are coming from we present two additional variations of the algorithm:

- **Non-removal contention-friendly version:** This version of the algorithm (Non-removal) does not perform any physical removals. A node that is deleted is marked as deleted, but stays in the skip list forever. This algorithm helps us examine the cost of contention caused

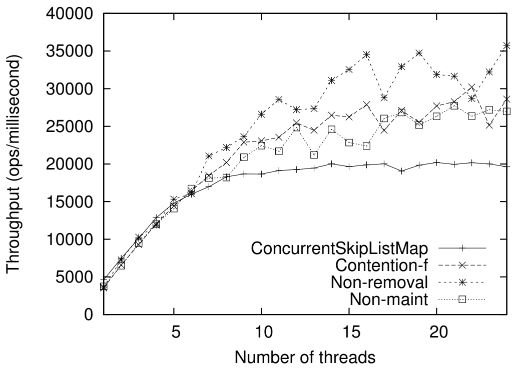


(a) Throughput, range of 128k elements (b) Slowdown, range of 128k elements

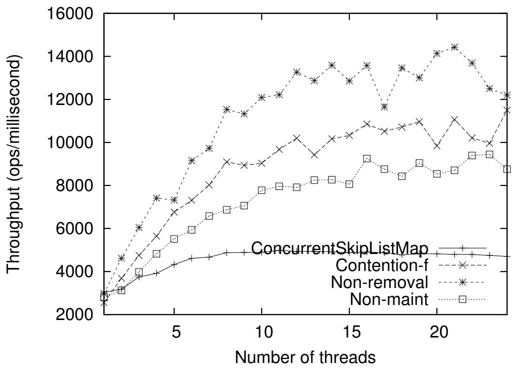


(c) Throughput, range of 128 million elements (d) Slowdown, range of 128 million elements

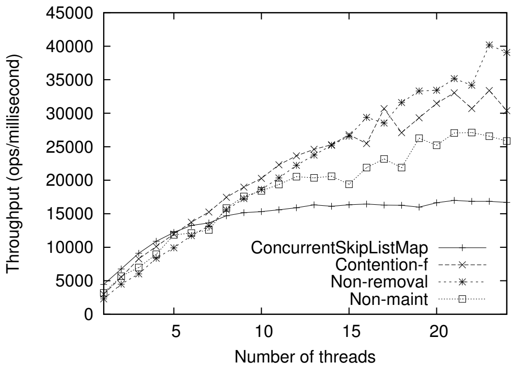
Figure 8.5: Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements



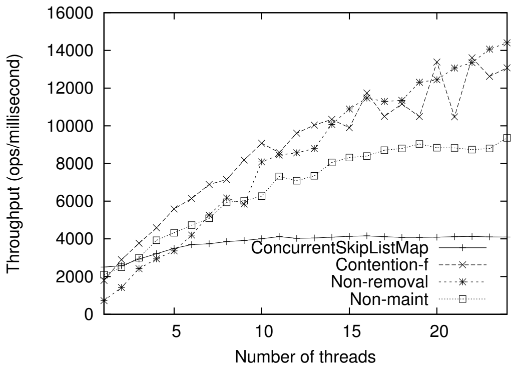
(a) 10% update, range of 128 elements



(b) 100% update, range of 128 elements

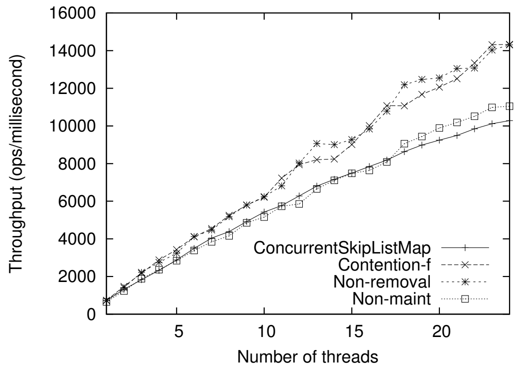


(c) 10% update, range of 128k elements

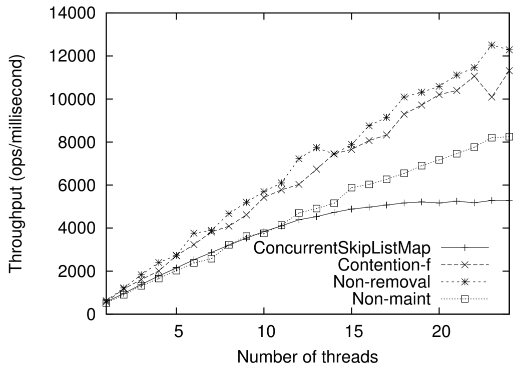


(d) 100% update, range of 128k elements

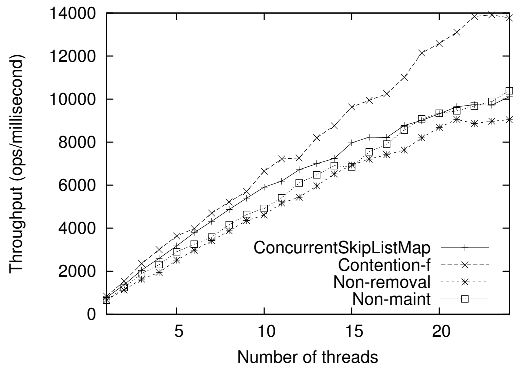
Figure 8.6: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements



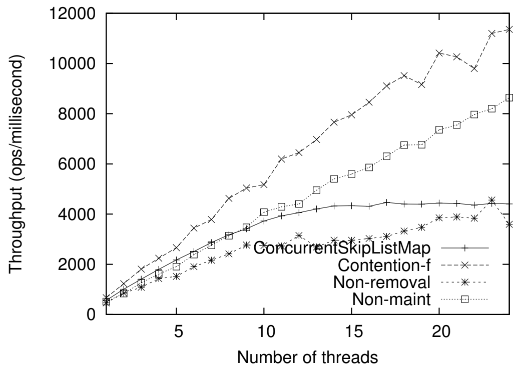
(a) 10% update, range of 128k elements



(b) 100% update, range of 128k elements

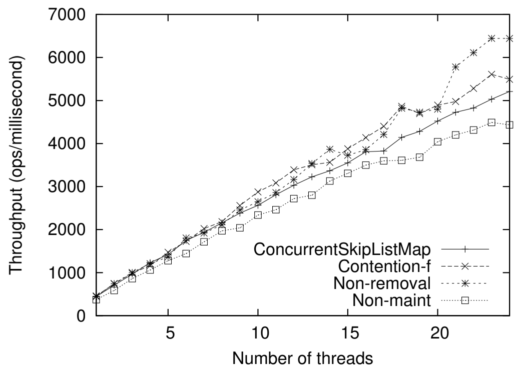


(c) 10% update, range of 128 million elements

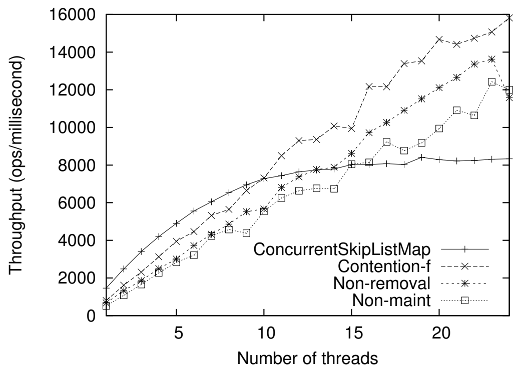


(d) 100% update, range of 128 million elements

Figure 8.7: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements



(a) Grow benchmark



(b) Shrink benchmark

Figure 8.8: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using the grow and shrink benchmark

by physical removals. A dedicated maintenance thread that takes care of modifications to the upper list levels.

- **Non-maintenance contention-friendly version:** This version of the algorithm (Non-maint) has no dedicated maintenance thread. It only uses the selective removals concept of contention friendliness; only nodes of height 1 are physically removed. This version helps us examine the benefits of using a maintenance thread. Given that there is no maintenance thread, modifications to the upper list levels are done as part of the abstract operations. A node's height, like in a traditional skip list, is chosen during the insert operation by a random function with the one exception that a height greater than 1 will only be chosen if both node's neighbors have a height of 1. This helps prevent there from being "too many" tall nodes due to the fact that only nodes of height 1 are physically removed.

Figure 8.4(a)-8.4(d) compares the effect of increasing the amount of update operations on the algorithms. The update ratio start at 0% and is increased to 100% (50% effective). The set contains approximately 64 elements throughout the benchmarks with small variations due to concurrency. The range of elements the abstract operations can choose from is either 128 or 128 thousand. The smaller range allows for higher contention on specific keys in the set while the larger range allows for more variation in the keys in the set. The graphs show both higher performance and less slowdown of the contention friendly algorithms compared to ConcurrentSkipListMap. Between the contention-friendly versions we see that Non-removal provides the best performance. This can be explained by the fact that since the size of the set is so small performing marked removals is much less contention-effective than physically removing nodes.

Figure 8.5(a)-8.5(d) is the same benchmark as Figure 8.4(a)-8.4(d) except it is run with a set size of approximately 64 thousand elements using ranges of 128 thousand and 128 million. Contention-f shows both good performance and small slowdown while Non-maint shows performance in-between Contention-f and ConcurrentSkipListMap. Non-removal shows the best performance with the range of 128 thousand elements, but performs poorly when the range is set to 128 million elements. Since the range is so large and Non-removal does not perform any physical removals the number of marked deleted nodes in the skip list grows so large that the cost of traversal becomes more expensive than in the other algorithms. In this benchmark we see the number of nodes in the skip list to be as large as 6 million.

Figure 8.6(a)-8.6(d) tests the scalability of the algorithms by showing the effect of increasing the number of threads from 1 to 24. The tests were done with a 10% and 100% update ratios with a set size of approximately 64 elements. Here we see better scalability from the contention-friendly algorithms compared to the ConcurrentSkipListMap. Non-removal shows in general the best performance due to the reduced contention from not doing physical removals (thanks especially to the small set size), followed by Contention-f, Non-maint, and finally ConcurrentSkipListMap.

Figure 8.7(a)-8.7(d) also tests the scalability of the algorithms and is the same benchmark as 8.6(a)-8.6(d) except it is run with a set size of approximately 64 thousand elements using ranges of 128 thousand and 128 million. At 10% update all algorithms show good scalability, while at 100% updates ConcurrentSkipListMap does not scale as well, with non-removal scaling the worst in the larger range benchmark due to it not physically removing nodes. In general Contention-f is the best performer followed by Non-maint.

The purpose of figure 8.8(a)-8.8(b) is to test the scalability of the algorithms when the number of elements in the set changes by a large amount. In the grow benchmark the size of the set starts at 0 elements and grows until a size of 500 thousand elements, while the shrink benchmark starts with a set of size 500 thousand elements and ends with 2,500 elements. Both benchmarks are executed with a 50% update ratio. All algorithms show good scalability in the grow benchmark with a small performance advantage going to the algorithms with maintenance threads (Contention-f, Non-removal) thanks to not requiring synchronization operations to the towers. In the shrink benchmark we see that the ConcurrentSkipListMap performs best at small thread counts while the contention-friendly algorithms show better scalability. Due to the decreasing list size Contention-f calls the lower-index-level procedure on average 4 times per run of the benchmark and at the end of the benchmark the skip list contains around 15 thousand marked deleted nodes. lower-index-level is called by the maintenance thread when it discovers that there are at least 10 times more marked deleted nodes than non-marked deleted ones. This number can be tuned so that the procedure is called more often.

8.11 Correctness

Here we show that the CF non-blocking skip list implements a map that is linearizable [135]. The proof is separated into two parts, first we show that performing contains, insert, and delete always results in a *valid* skip list structure. Second we show that each operation has a linearization point.

Definitions. The skip list presented here represents the set (or map) abstract data type. A key k is in the set if there is a path from the field *top* to a node with key equal to k with a non-$\perp$ value, otherwise it is not in the set. Therefore a *valid* skip list has the following properties: (i) the nodes in the skip list are sorted by their keys in ascending order, (ii) there is a path from the field *top* to at most one node with a key k at any point in time and (iii) every node with value $v \neq node$ has a path to it from *top*. We consider that an operation contains, insert, and delete is a *success* if it returns true, otherwise it *fails*.

For the sake of simplicity the structure is initialized with a single node with key $-\infty$ and a tower of maximum height. Each of the IndexItems of the tower have their *right* pointer initialized to $\perp$ and the node's *next* pointer is also initialized to $\perp$.

Before we define the linearization points of the operations we need the following two lemmas.

Lemma 40 *If a node n is such that $n.next = n'$ where $n'.marker =$ true and $n.value = n$ then there is no longer a path from top to n.*

Proof.[Proof sketch] Nodes are only unlinked from the list during a help-remove operation. To prove the lemma it needs to be shown that the only two nodes unlinked from the list are *node* and the marker node that follows it in the list. The CAS on line 117 ensure that there are no nodes in between *node* and its predecessor. Lines 112-113 CAS exactly one marker node after *node*. To show that no nodes are added before or after the marker node we will show that pointers to and from marker nodes are never modified by considering all locations where the structure of the list is modified. First we consider the places where a new node can be added to the list. There are two places were a new node can be added to the list, this is on line 113 of the help-remove procedure and line 45 of the insert operation. In the case of the help-remove procedure, before adding a new marker node it checks that the predecessor and the successor nodes are not markers (line

106 and 109). The same is done during the insert operation on lines 87 and 92 of get-next-node and 44 of insert. The only other place where the list structure is modified is on line 117 of the help-remove procedure, but line 115 checks that the pointer modified is not from a marker node. □

Lemma 41 *A successful* remove *operation on a node of a valid skip list results in a valid skip list with the node physically removed from the list (i.e. no path from top to the node exists). The state of the elements in the abstraction is left unchanged.*

Proof.[Proof sketch] The operation starts by performing a CAS on the v field of the node, changing it from $\perp$ to point to the node. Atomically changing the value from $\perp$ ensures that the key of this node was not in the set when the removal starts. If this CAS succeeds then the help-remove procedure is called. This procedure starts by ensuring a marker node is the following node in the list (line 109). If not then such a node is allocated and added to the list using a CAS (lines 112-113). The CAS ensues the pointer has not changed since it was first read on line 108 or 114 so that no newly inserted nodes are lost. The last modification done by the removal operation is the CAS done on line 117 which unlinks *node* and its successor (the marker node) from the list ensuring. To finish showing that that the removal does not modify the set it needs to be shown that only these two nodes are unlinked from the list by this CAS, but this is ensured by lemma 40. □

Lemma 42 *A* contains *operation performed on a valid skip list is linearizable and results in a valid skip list.*

Proof.[Proof sketch] The contains operation does not modify the skip list so it always results in a valid list.

Success. A successful contains operation means that an element with key k exists in the set. Therefore the following must be true at its linearization point: There exists a node n with key k, $v \neq \perp$, and $v \neq n$. The linearization point for this is line 68 where the operation reads the v field. From the previous line (67) it knows that the node's key is equal to k and the checks on line 69 ensure $v \neq \perp$ and $v \neq n$.

Failure. A successful contains operation means that an element with key k does not exist in the set. There are two cases:

1. The operation finds no node with key k and returns false. This means that the check of the key on line 67 must have failed. Now the following must be true at the operation's linearization point: There does not exist a node n with key k and $v \neq n$ in the list. The linearization point is line 88 of get-next-node where the *next* pointer of *node* is read. First notice that the operation never traverses past a node with key larger then k (line 75 of get-next-index and line 92 of get-next-node). Therefore this *node* has a smaller key then k and must be in the list by line 87 of get-next-node and lemma 40. Also given that the list is valid the nodes are sorted by their keys in ascending order and that the next node in the list has a larger key then k (by line 92 of get-next-node) so there exists no node with key k in the list.

2. The operation finds a node in the list with key k that has been marked deleted. This means that the check of the key on line 67 must have succeeded. Now the following must be

true at the operation's linearization point: There exists a node n with key k and $v = \perp$. The linearization point for this is line 87 of get-next-node where the node is seen to have $v \neq node$. Given that the list is valid, there exists no other node with key k in the list and since the v field of the node is neither *node* (line 87) nor not equal to $\perp$ (line 69) then it must be $\perp$ at the linearization point and k does not exist in the set.

□

Lemma 43 *An* insert *operation performed on a valid list is linearizable and results in a valid list.*

Proof.[Proof sketch] **Success.** A successful insert operation means that an element with key k was added to the set. There are two cases:

1. The operation found a node with key k that was marked deleted. In this case following must be true before the linearization point: There exists a node with key k and $v = \perp$. And after the linearization point: There exists exactly one node with key k, $v \neq \perp$, $v \neq node$, and $v \neq node$. The linearization point for this is when the value of the node is CASed from $\perp$ to v on line 35. This CAS ensures the precondition because of the validation (that checks that the node's key is k and value is $\perp$) done on lines 87 and 92 of get-next-node and line 34 of insert must be valid for the CAS to succeed. This CAS (line 35) then also produces the post condition by updating the node's value to the value that was given as input. In this case no modifications are made to the structure of the list or to any nodes with $v = node$ so the resulting list must still be *valid*.

2. The operation found no node with key k in the list. In this case following must be true before the linearization point: There does not exist a node with key k, and $v \neq node$. And after the linearization point: There exists exactly one node that has a path to from *top* with key k, $v \neq \perp$, and $v \neq node$. The linearization point is when the newly allocated node is linked into the list by changing the predecessor p's *next* pointer to point to the new node (line 45 using a CAS. The following checks ensure that the node is inserted in the correct location in the list (ensuring the sorted property of the *valid* list) and that it is not inserted before or after a marker node which ensures by lemma 40 that the predecessor is in the list. Line 87 of get-next-node ensures that the predecessor is not a marker $p.v \neq p$, line 92 of get-next-node ensures that the successor has key greater than k and predecessor has key smaller than k, and line 44 of insert ensures that the successor is not a marker by checking that its value is not a pointer to itself. Given that a node's key never changes and that a node that is allocated as a marker always stays a marker, the CAS ensures that the predecessor and the successor are the same nodes and the checks are still valid. Within the setup_node operation a new node is allocated, has its key, value and pointers set (lines 59-61). This setup followed by the CAS ensures the post condition.

Failure. A failed insert operation follows the same structure as a successful contains operation.

□

Lemma 44 *A* delete *operation performed on a valid list is linearizable and results in a valid list.*

Proof.[Proof sketch] **Success.** A successful delete operation means that an element with key k was removed from the the set. This means that the operation found a node with key k that was not marked deleted. In this case following must be true before the linearization point: There exists a node with key k, $v \neq \perp$, and $v \neq node$. And after the linearization point: There exists exactly one node with key k and $v = \perp$. The linearization point for this is when the value of the node is changed to $\perp$ by the CAS on line 22. This CAS ensures the precondition because of the validation (that checks that the node's key is k and value is neither $\perp$ nor *node*) done on lines 87 and 92 of get-next-node and line 21 of delete must be valid for the CAS to succeed. This CAS (line 22) also produces the post condition by changing the nodes value to $\perp$. In this case no modifications are made to the structure of the list or to any nodes with $v = node$ so the resulting list must still be *valid*.

Failure. A failed delete operation follows the same structure as a failed contains operation.
□

Algorithm 11 Contention-friendly non-blocking skip list – abstract operations by process p

```
State of node:
  node a record with fields:
    $k \in \mathbb{N}$, the node key
    $v$, the node's value, a value of $\bot$ indicates
      the node is logically deleted
    $marker \in \{\mathsf{true}, \mathsf{false}\}$, indicates if this is
      a marker node
    next, pointer to the next node in the list
    prev, pointer to the previous node in the list
    level, integer indicating the level of the node,
      initialized to 0

delete$(k)_p$:
  $node \leftarrow$ traverse-tower$(top, k)$
  while true do
    $node \leftarrow$ get-next-node$(node, k)$
    if $node.k \neq k$ then
      $result \leftarrow$ false
      break()
    else
      $v \leftarrow node.v$
      if $(v \neq \bot \wedge v \neq node)$ then
        if CAS$(node.v, v, \bot)$ then // compare-and-swap
          $result \leftarrow$ true
          break()
      else
        $result \leftarrow$ false
        break()
  return $result$

insert$(k, v)_p$:
  $node \leftarrow$ traverse-tower$(top, k)$
  while true do
    $node \leftarrow$ get-next-node$(node, k)$
    if $node.k = k$ then
      if $node.v = \bot$ then // logical insertion
        if CAS$(node.v, \bot, v)$ then // compare-and-swap
          $result \leftarrow$ true
          break()
      else
        $result \leftarrow$ false
        break()
    else
      $new \leftarrow$ setup_node$(node, k, v)$
      $next \leftarrow new.next$
      if $next.val \neq next$ then // logical insertion
        if CAS$(node.next, next, new)$ then // compare-and-swap
          $next.prev \leftarrow new$
          $result \leftarrow$ true
          break()
  return $result$

State of index-item:
  item a record with fields:
    right, pointer to the next
      item in the SkipList
    down, pointer to the IndexItem
      one level below in the SkipList
    node, pointer a node in the list
      at the bottom of the SkipList

setup-node$(node, k, v)_p$:
  $new.k \leftarrow k$; $new.v \leftarrow v$ // allocate a node called new
  $new.prev \leftarrow node$
  $new.next \leftarrow node.next$
  return $new$

contains$(k)_p$:
  $item \leftarrow$ traverse-tower$(top, k)$ // find the right tower
  $node \leftarrow$ get-next-node$(item, k)$ // find the right node
  $result \leftarrow$ false
  if $node.k = k$ then
    $v \leftarrow node.v$
    if $(v \neq \bot \wedge v \neq node)$ then
      $result \leftarrow$ true
  return $result$

traverse-tower$(item, k)_p$:
  while true do
    $nitem \leftarrow item.right$
    if $nitem.node.k > k$ then
      $nitem \leftarrow item.down$
      if $nitem = \bot$ then
        $result \leftarrow item.node$
        break()
    else if $nitem.node.k = k$ then
      $result \leftarrow item.node$
      break()
    $item \leftarrow nitem$
  return $result$

get-next-node$(node, k)_s$:
  while true do
    while $node.v = node$ do $node \leftarrow node.prev$ // backtack
    $next \leftarrow node.next$
    if $(next \neq \bot \wedge next.v = next)$ then
      help-remove$(node, next)$ // help the removal
      $next \leftarrow node.next$
    if $(next = \bot \vee next.k > k)$ then
      $result \leftarrow node$
      break()
    $node \leftarrow next$
  return $result$
```

Algorithm 12 Contention-friendly non-blocking skip list – structural adaptation by process p

```
remove($pred, node$)$_p$:
  $result \leftarrow$ false
  if $node.level = 0$ then
    CAS($node.v, \bot, node$) // compare-and-swap
    if $node.v = node$ then
      help_remove($pred, node$)
      $result \leftarrow$ true
  return $result$

help-remove($pred, node$)$_p$:
  if ($node.val \neq node \vee node.marker$) then
    return
  $n \leftarrow node.next$
  while $\neg n.marker$ do
    $new \leftarrow$ setup_node($node, \bot, \bot$)
    $new.v \leftarrow new$
    $new.marker \leftarrow$ true
    CAS($node.next, n, new$) // compare-and-swap
    $n \leftarrow node.next$
  if ($pred.next \neq node \vee pred.marker$) then
    return
  CAS($pred.next, node, n.next$) // compare-and-swap

lower-index-level()$_p$:
  $index \leftarrow first[2].next$
  while $index \neq \bot$ do
    $index.down \leftarrow \bot$
    $index.node.height \leftarrow index.node.height - 1$
    $index \leftarrow index.next$
  // Update the index of the first array

raise-index-level($i$)$_p$:
  $prev\text{-}tall \leftarrow first[i+1]$
  $index \leftarrow first[i]$
  while true do
    $next \leftarrow index.right$
    if $next = \bot$ then
      break()
    $prev \leftarrow index.prev$
    if ($prev.node.level \leq i$
      $\wedge\ index.node.level \leq i$
        $\wedge\ next.node.level \leq i$) then
      // Allocate a new index-item called new
      $new.down \leftarrow index$
      $new.node \leftarrow index.node$
      $new.right \leftarrow prev\text{-}tall.right$
      $prev\text{-}tall.right \leftarrow new$
      $index.node.level \leftarrow i+1$
      $prev\text{-}tall \leftarrow new$
    $index \leftarrow index.right$
```

Chapter 9

A Contention Friendly Hash-Table

9.1 Lock Free Algorithm

9.2 Transactional Memory Based Algorithm

Conclusion

Bibliography

[1] Attiya H. and Hillel E., Single-version STM Can be Multi-version Permissive. *Proc. 12th Int'l Conference on Distributed Computing and Networking (ICDCN'11)*, Springer-Verlag, LNCS #6522, pp. 83-94, 2011.

[2] Babaoğlu Ö. and Marzullo K., Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Chapter 4 in "Distributed Systems". ACM Press, Frontier Series, pp 55-93, 1993.

[3] Bernstein Ph.A., Shipman D.W. and Wong W.S., Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, SE-5(3):203-216, 1979.

[4] Cachopo J. and Rito-Silva A., Versioned Boxes as the Basis for Transactional Memory. *Science of Computer Progr.*, 63(2):172-175, 2006.

[5] Crain T., Imbs D. and Raynal M., Read invisibility, virtual world consistency and permissiveness are compatible. *Tech Report #1958*, IRISA, Univ. de Rennes 1, France, November 2010.

[6] Dice D., Shalev O. and Shavit N., Transactional Locking II. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag, LNCS #4167, pp. 194-208, 2006.

[7] Felber P., Fetzer Ch., Guerraoui R. and Harris T., Transactions are coming Back, but Are They The Same? *ACM Sigact News, DC Column*, 39(1):48-58, 2008.

[8] Guerraoui R., Henzinger T.A., Singh V., Permissiveness in Transactional Memories. *Proc. 22th Int'l Symposium on Distributed Computing (DISC'08)*, Springer-Verlag, LNCS #5218, pp. 305-318, 2008.

[9] Guerraoui R. and Kapałka M., On the Correctness of Transactional Memory. *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, ACM Press, pp. 175-184, 2008.

[10] Harris T., Cristal A., Unsal O.S., Ayguade E., Gagliardi F., Smith B. and Valero M., Transactional Memory: an Overview. *IEEE Micro*, 27(3):8-29, 2007.

[11] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News, DC Column*, 39(1): 62-72, 2008.

[12] Herlihy M.P. and Moss J.E.B., Transactional Memory: Architectural Support for Lock-free Data Structures. *Proc. 20th ACM Int'l Symposium on Computer Archictecture (ISCA'93)*, pp. 289-300, 1993.

[13] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[14] Imbs D. and Raynal M., A Lock-based STM Protocol that Satisfies Opacity and Progressiveness. *12th Int'l Conference On Principles Of Distributed Systems (OPODIS'08)*, Springer-Verlag LNCS #5401, pp. 226-245, 2008.

[15] Imbs D. and Raynal M., Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort. *10th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer-Verlag LNCS #5408, pp. 67-78, January 2009.

[16] Imbs D. and Raynal M., A versatile STM protocol with Invisible Read Operations that Satisfies the Virtual World Consistency Condition. *16th Colloquium on Structural Information and Communication Complexity (SIROCCO'09)*, Springer Verlag LNCS, #5869, pp. 266-280, 2009.

[17] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, 1978.

[18] Marathe V.J., Spear M.F., Heriot C., Acharya A., Eisentatt D., Scherer III W.N. and Scott M.L., Lowering the Overhead of Software Transactional Memory. *Proc 1rt ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.

[19] Papadimitriou Ch.H., The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4):631-653, 1979.

[20] Perelman D., Fan R. and Keidar I., On Maintaining Multiple versions in STM. *Proc. 29th annual ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 16-25, 2010.

[21] Riegel T., Fetzer C. and Felber P., Time-based Transactional Memory with Scalable Time Bases. *Proc. 19th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, ACM Press, pp. 221-228, 2007.

[22] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2):99-116, 1997.

[23] Schwarz R. and Mattern F., Detecting Causal Relationship in Distributed Computations: in Search of the Holy Grail. *Distributed Computing*, 7:149-174, 1993.

[24] Afek Y., Dauber D. and Touitou D., Wai-free made Fast. *Proc. 7th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, ACM Press, pp. 538-547, 1995.

[25] Anderson J. and Moir M., Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317-1332, 1999.

[26] Ansar M., Luján M., Kotselidis Ch., Jarvis K., Kirkham Ch. and Watson Y., Steal-on-abort: Dynamic Transaction Reordering to Reduce Conflicts in Transactional Memory. *4th Int'l ACM Sigplan Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09)*, ACM Press, pp. 4-18, 2009.

[27] Attiya H. and Milani A., Transactional Scheduling for Read-Dominated Workloads. *13th Int'l Conference on Principles of Distributed Systems (OPODIS'09)*, Springer Verlag LNCS #5923, pp. 3-17, 2009.

[28] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[29] Chuong Ph., Ellen F. and Ramachandran V., A Universal Construction for Wait-free Transaction Friendly Data Structures. *Proc. 22th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, ACM Press, pp. 335-344, 2010.

[30] Crain T., Imbs D. and Raynal M., Towards a universal construction for transaction-based multiprocess programs. *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN 2012)*, Springer-Verlag LNCS, 2012.

[31] Dijkstra E.W.D., Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):69, 1968.

[32] Dragojević A., Guerraoui R. and Kapałka M., Stretching Transactional Memory. *Proc. Int'l 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*, ACM Press, pp. 155-165, 2009.

[33] Fatourou P. and Kallimanis N., The Red-blue Adaptive Universal Construction. *Proc. 22nd Int'l Symposium on Distributed Computing (DISC '09)*, Springer-Verlag, LNCS#5805, pp. 127-141, 2009.

[34] Felber P., Compiler Support for STM Systems. Lecture given at the *TRANSFORM Initial Training School*, University of Rennes 1 (France), 7-11 February 2011.

[35] Felber P., Fetzer Ch. and Riegel T., Dynamic Performance Tuning of Word-Based Software Transactional Memory. *Proc. 13th Int'l ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)* , ACM Press,pp. 237-246, 2008.

[36] Felber P., Fetzer Ch., Guerraoui R. and Harris T., Transactions are Coming Back, but Are They The Same? *ACM Sigact News, Distributed Computing Column*, 39(1):48-58, 2008.

[37] Frølund S. and Guerraoui R., X-Ability: a Theory of Replication. *Distributed Computing*, 14(4):231-249, 2001.

[38] Guerraoui R., Henzinger Th. A. and Singh V., Permissiveness in Transactional Memories. *Proc. 22nd Int'l Symposium on Distributed Computing (DISC '08)* , Springer-Verlag, LNCS#5218, pp. 305-319, 2008.

[39] Guerraoui R., Herlihy M. and Pochon B., Towards a Theory of Transactional Contention Managers. *Proc. 24th Int'l ACM Symposium on Principles of Distributed Computing (PODC'05)*, ACM Press, pp. 258-264, 2005.

[40] Guerraoui R., Kapałka M. and Kouznetsov P., The Weakest Failure Detectors to Boost Obstruction-freedom. *Distributed Computing*, 20(6):415-433, 2008.

[41] Guerraoui R. and Kapałka M., Principles of Transactional Memory. *Synthesis Lectures on Distributed Computing Theory*, Morgan & Claypool Publishers, 180 pages, 2010.

[42] Harris T., Cristal A., Unsal O.S., Ayguade E., Gagliardi F., Smith B. and Valero M., Transactional Memory: an Overview. *IEEE Micro*, 27(3):8-29, 2007.

[43] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[44] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News*, 39(1): 62-72, 2008.

[45] Herlihy M., Luchangco V., Moir M. and Scherer III W.M., Software Transactional Memory for Dynamic-Sized Data Structures. *Proc. 22nd Int'l ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 92-101, 2003.

[46] Herlihy M.P. and Moss J.E.B., Transactional Memory: Architectural Support for Lock-free Data Structures. *Proc. 20th ACM Int'l Symposium on Computer Architecture (ISCA'93)*, ACM Press, pp. 289-300, 1993.

[47] Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming. *Morgan Kaufmann Pub.*, San Francisco (CA), 508 pages, 2008.

[48] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[49] Hewitt C.E. and Atkinson R.R., Specification and Proof Techniques for Serializers. *IEEE Transactions on Software Engineering*, SE5(1):1-21, 1979.

[50] Hoare C.A.R., Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549-557, 1974.

[51] Larus J. and Kozyrakis Ch., Transactional Memory: Is TM the Answer for Improving Parallel Programming? *Communications of the ACM*, 51(7):80-89, 2008.

[52] Maldonado W., Marlier P., Felber P., Lawall J., Muller G. and Revière E., Deadline-Aware Scheduling for Software Transactional Memory. *41th IEEE/IFIP Int'l Conference on Dependable Systems and Networks 'DSN'11)*, IEEE CPS Press, June 2011.

[53] Michael M.M. and Scott M.L., Simple, Fast and Practical Blocking and Non-Blocking Concurrent Queue Algorithms. *Proc. 15th Int'l ACM Symposium on Principles of Distributed Computing (PODC'96)*, ACM Press, pp. 267-275, 1996.

[54] Raynal M., Synchronization is Coming Back, But Is It the Same? Keynote Speech. *IEEE 22nd Int'l Conference on Advanced Information Networking and Applications (AINA'08)*, pp. 1-10, 2008.

[55] Rosenkrantz D.J., Stearns R.E. and Lewis Ph.M., System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3(2): 178-198, 1978.

[56] Spear M.F., Silverman M., Dalessandro L., Michael M.M. and Scott M.L., Implementing and Exploiting Inevitability in Software Transactional Memory. *Proc. 37th Int'l Conference on Parallel Processing (ICPP'08)*, IEEE Press, 2008.

[57] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2):99-116, 1997.

[58] Wamhoff J.-T. and Fetzer Ch., The Universal Transactional Memory Construction. *Tech Report*, 12 pages, University of Dresden (Germany), 2010.

[59] Wamhoff J.-T., Riegel T., Fetzer Ch. and Felber F., RobuSTM: A Robust Software Transactional Memory. *Proc. 12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer Verlag LNCS #6366, pp. 388-404, 2010.

[60] Welc A., Saha B. and Adl-Tabatabai A.-R., Irrevocable Transactions and their Applications. *Proc. 20th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*, ACM Press, pp. 285-296, 2008.

[61] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proc. of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[62] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2009.

[63] Lucia Ballard. Conflict avoidance: Data structures in transactional memory, May 2006. Undergraduate thesis, Brown University.

[64] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica 1*, 1(4):290–306, 1972.

[65] Luc Bougé, Joaquim Gabarro, Xavier Messeguer, and Nicolas Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries, 1998. Research Report 1998-18, ENS Lyon.

[66] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[67] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE Int'l Symp. on Workload Characterization*, 2008.

[68] Christopher Cole and Maurice Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58(3):310–324, 2005.

[69] Luke Dalessandro, Michael Spear, and Michael L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[70] D. Dice, O. Shalev, , and N. Shavit. Transactional locking II. In *Proc. of the 20th Int'l Symp. on Distributed Computing*, 2006.

[71] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[72] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[73] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2008.

[74] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *Proc. of the 23rd Int'l Symp. on Distributed Computing*, 2009.

[75] Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. In *Proc. of the ACM/IFIP/USENIX 12th Int'l Middleware Conference*, 2011.

[76] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual Symp. on Foundations of Computer Science*, 1978.

[77] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2005.

[78] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture*, 1993.

[79] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2008.

[80] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd Annual ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 2003.

[81] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009.

[82] J. L. W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26:895–901, 1983.

[83] Udi Manbar and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.*, 9(3):439–455, 1984.

[84] C. Mohan. Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In *Proc. of the 16th Int'l Conference on Very Large Data Bases*, 1990.

[85] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, 2006.

[86] Yang Ni, Vijay Menon, Ali-Reza Abd-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2007.

[87] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proc. of the 10th ACM Symp. on Principles of Database Systems*, 1991.

[88] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proc. of the 6th ACM Symp. on Principles of Database Systems*, 1987.

[89] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proc. of the 23rd ACM Symp. on Parallelism in Algorithms and Architectures*, 2011.

[90] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[91] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[92] Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, 1995.

[93] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

[94] L. Ballard. Conflict avoidance: Data structures in transactional memory, May 2006. Undergraduate thesis, Brown University.

[95] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.

[96] L. Bougé, J. Gabarro, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report RR1998-18, ENS Lyon, 1998.

[97] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[98] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012.

[99] D. Dice, O. Shalev, , and N. Shavit. Transactional locking II. In *Proc. of the 20th Int'l Symp. on Distributed Computing*, 2006.

[100] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44:779–805, November 1997.

[101] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[102] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proc. of the 23rd Int'l Symp. on Distributed Computing*, 2009.

[103] M. Fomitchev and E. 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.

[104] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University, September 2003.

[105] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285 –297, 1999.

[106] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual Symp. on Foundations of Computer Science*, 1978.

[107] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[108] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Par. Prog.*, 2008.

[109] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[110] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kauffman, February 2008.

[111] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[112] J. L. W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26:895–901, 1983.

[113] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

[114] U. Manbar and R. E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.*, 9(3):439–455, 1984.

[115] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *SC*, pages 1–11, 2010.

[116] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[117] C. Mohan. Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In *Proc. of the 16th Int'l Conference on Very Large Data Bases*, 1990.

[118] Y. Ni, V. Menon, A.-R. Abd-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2007.

[119] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proc. of the 10th ACM Symp. on Principles of Database Systems*, 1991.

[120] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proc. of the 6th ACM Symp. on Principles of Database Systems*, 1987.

[121] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33, June 1990.

[122] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.

[123] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[124] H. Sutter. Choose concurrency-friendly data structures. *Dr. Dobb's Journal*, June 2008.

[125] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976. 10.1007/BF01683268.

[126] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[127] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.

[128] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, 2012.

[129] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *PODC*, pages 190–199, 2001.

[130] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[131] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.

[132] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, 2004.

[133] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University, September 2003.

[134] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[135] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[136] D. Lea. Jsr-166 specification request group.

[137] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[138] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS*, 1987.

[139] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33, June 1990.

[140] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[141] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC*, pages 1438–1445, 2004.

[142] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *DISC*, pages 157–171, 2009.

[143] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

[144] Afek, Y., Avni, H., Dice, D., Shavit, N.: Efficient lock free privatization. In: *Proc. 14th Int'l conference on Principles of Distributed Systems (OPODIS'10)*, pp. 333–347, Springer-Verlag, LNCS #6490 (2010)

[145] Crain, T., Imbs, D., Raynal, M.: Towards a Universal Construction for Transaction-based Multiprocess Programs. In: *Proc. 13th Int'l Conference on Distributed Computing and Networking (ICDCN'12)*, pp. 61–75, Springer Verlag LNCS #7129 (2012)

[146] Dalessandro, L., Scott, M.: Strong Isolation is a Weak Idea. In: *Proc. Workshop on transactional memory (TRANSACT'09)* (2009)

[147] Dice, D., Matveev, A., Shavit, N.: Implicit privatization using private transactions. In: *Proc. Workshop on transactional memory (TRANSACT'10)* (2010)

[148] Dice, D., Shalev O., Shavit, N.: Transactional Locking II. In: *Proc. 20th Int'l Symp. on Distributed Computing (DISC'06)*, Springer-Verlag, LNCS #4167, pp. 194–208 (2006)

[149] Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, ACM Press, pp. 175–184 (2008)

[150] Harris, T., Larus, J., Rajwar, R.: Transactional Memory, *2nd edition, Synthesis Lectures on Computer Architecture*, Morgan & Claypool Publishers (2006)

[151] Herlihy, M., Moss J.M.B.: Transactional memory: architectural support for lock-free data structures, In: *Proc. of the 20th annual Int'l Symposium on Computer Architecture (ISCA '93)*, ACM Press, pp. 289–300 (1993)

[152] Imbs, D., Raynal, M.: A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition. In: *16th Colloquium on Structural Information and Communication Complexity (SIROCCO'09)*, Springer Verlag LNCS #5869, pp. 266–280 (2009)

[153] Maessen, J.-W., Arvind, M.: Store Atomicity for Transactional Memory. *Electronic Notes on Theoretical Computer Science*, 174(9):117–137 (2007).

[154] Martin, M., Blundell, C., Lewis, E.: Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2): (2006)

[155] Matveev, A., Shavit, N.: Towards a Fully Pessimistic STM Model. In: *Proc. Workshop on transactional memory (TRANSACT'12)* (2012)

[156] Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: *SIGARCH Comput. Archit. News*, 35(2):69–80 (2007)

[157] Papadimitriou, Ch.H.: The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4):631–653 (1979)

[158] Schneider, F., Menon, V., Shpeisman, T., Adl-Tabatabai, A.: Dynamic optimization for efficient strong atomicity. In: *ACM SIGPLAN Noticers*, 43(10):181–194 (2008)

[159] Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Delaunay Triangulation with Transactions and Barriers. In: *Proc. 10th IEEE Int'l Symposium on Workload Characterization (IISWC '07)*, IEEE Computer Society, pp. 107–113 (2007)

[160] Shavit, N., Touitou, D.: Software transactional memory. Software Transactional Memory. In: *Distributed Computing*, 10(2):99–116 (1997)

[161] Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in STM. In: *ACM SIGPLAN Noticers*, 42(6):78–88 (2007)

[162] Spear M.F., Dalessandro L., Marathe V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: *Proc 12th Int'l Conf. on Principles of Distributed Systems (OPODIS '08)*, Springer-Verlag LNCS #5401, pp. 275–294 (2008)

[163] Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. In: *Proc. 26th annual ACM symposium on Principles of Distributed Computing (PODC '07)*, . ACM press, pp. 338–339, (2007)

List of Publications

International Conferences Articles

Technical Reports