Designing, Modeling, and Optimizing Transactional Data Structures

Ahmed Hassan

Dissertation Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

September 1, 2015
Blacksburg, Virginia

Keywords: Transactional Memory, Transactional Boosting, Data Structures, Hybrid Transactions, Semantic Validation

Designing, Modeling, and Optimizing Transactional Data Structures

Ahmed Hassan

(ABSTRACT)

Transactional memory (TM) has emerged as a promising synchronization abstraction for multi-core architectures. Unlike traditional lock-based approaches, TM shifts the burden of implementing threads synchronization from the programmer to an underlying framework using hardware (HTM) and/or software (STM) components.

Although TM can be leveraged to implement *transactional data structures* (i.e., those where multiple operations are allowed to execute atomically, all-or-nothing, according to the transaction paradigm), its intensive speculation may result in significantly lower performance than the optimized concurrent data structures. This poor performance motivates the need to find other, more effective, alternatives for designing transactional data structures without losing the simple programming abstraction proposed by TM.

To do so, we identified three major challenges that need to be addressed to design efficient transactional data structures. The first challenge is *composability*, namely allowing an atomic execution of two or more data structure operations in the same way as TM provides, but without its high overheads. The second challenge is *integration*, which enables the execution of data structure operations within generic transactions that may contain other *memory-based* operations. The last challenge is *modeling*, which encompasses the necessity of defining a unified formal methodology to reason about the correctness of transactional data structures.

In this dissertation, we propose different approaches to address the above challenges. First, we address the *composability* challenge by introducing an optimistic methodology to efficiently convert concurrent data structures into transactional ones. Second, we address the *integration* challenge by injecting the semantic operations of those transactional data structure into TM frameworks, and by presenting two novel STM algorithms in order to enhance the overall performance of those frameworks. Finally, we address the *modeling* challenge by presenting two models for concurrent and transactional data structures designs.

- Our first main contribution in this dissertation is Optimistic transactional boosting (OTB), a methodology to design transactional versions of the highly concurrent optimistic (i.e., lazy) data structures. An earlier (pessimistic) boosting proposal added a layer of abstract locks on top of existing concurrent data structures. Instead, we propose an optimistic boosting methodology, which allows greater data structure-specific optimizations, easier integration with TM frameworks, and lower restrictions on the operations than the original (more pessimistic) boosting methodology.

  Based on the proposed OTB methodology, we implement the transactional version of two list-based data structures (i.e., set and priority queue). Then, we present *TxCF-Tree*, a balanced tree whose design is optimized to support transactional accesses. The core optimizations of TxCF-Tree's operations are: providing a traversal phase that does not use any lock and/or speculation and deferring the lock acquisition or physical modification to the transaction's commit phase; isolating the structural operations (such as re-balancing) in an interference-less housekeeping thread; and minimizing the interference between structural operations and the critical path of semantic operations (i.e., additions and removals on the tree).

- Our second main contribution is to integrate OTB with both STM and HTM algorithms. For STM, we extend the design of both DEUCE, a Java STM framework, and RSTM, a C++ STM framework, to support the integration with OTB. Using our extension, programmers can include both OTB data structure operations and traditional memory reads/writes in the same transaction. Results show that OTB performance is closer to the optimal lazy (non-transactional) data structures than the original boosting algorithm.

  On the HTM side, we introduce a methodology to inject semantic operations into the well-known hybrid transactional memory algorithms (e.g., HTM-GL, HyNOrec, and NOrecRH). In addition, we enhance the proposed semantically-enabled HTM algorithms with a lightweight adaptation mechanism that allows bypassing the HTM paths if the overhead of the semantic operations causes repeated HTM aborts. Experiments on micro- and macro-benchmarks confirm that our proposals outperform the other TM solutions in almost all the tested workloads.

- Our third main contribution is to enhance the performance of TM frameworks in general by introducing two novel STM algorithms. Remote Transaction Commit (RTC) is a mechanism for executing commit phases of STM transactions in dedicated server cores. RTC shows significant improvements compared to its corresponding validation based STM algorithm (up to 4x better) as it decreases the overhead of spin locking during commit, in terms of cache misses, blocking of lock holders, and CAS operations. Remote Invalidation (RInval) applies the same idea of RTC on invalidation based STM algorithms. Furthermore, it allows more concurrency by executing commit and invalidation routines concurrently in different servers. RInval performs up to 10x better than its corresponding invalidation based STM algorithm (InvalSTM), and up to 2x better than its corresponding validation-based algorithm (NOrec).

- Our fourth and final main contribution is to provide a theoretical model for concurrent and transactional data structures. We exploit the similarities of the OTB-based data structures and provide a unified model to reason about the correctness of those designs. Specifically, we extend a recent approach that models data structures with concurrent readers and a single writer (called *SWMR*), and we propose two novel models that additionally allow *multiple writers* and *transactional execution*. Those models are more practical because they cover a wider set of data structures than the original SWMR model.

This work is supported in part by US National Science Foundation under grants CNS-1116190, CNS-1217385, and Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

Dedication

To my parents, my wife, and my kids.

Acknowledgments

First and foremost, all praise is due to Allah, the One who by His blessing and favor, a perfect goodness is accomplished.

I would like to thank my advisor, Prof. Binoy Ravindran, for his endless help and support. In addition to his valuable technical help throughout my PhD journey, he always provided me with sincere advice and encouragement, and never lost trust in me. I would also like to express my gratefulness to my committee members: Prof. Robert P. Broadwater, Prof. Eli Telivich, Prof. Chao Wang, and Prof. Jules White, for the insightful comments and discussions.

I am so grateful to my colleagues in the *Systems Software Research Group* (SSRG), who formed the best environment for a PhD student to succeed. I would like to extend a special thanks to Dr. Roberto Palmieri, Dr. Sebastiano Peluso, and Dr. Mohamed Mohamedin, for the great collaboration. Working with such a team greatly benefited my academic career and personal life. I will never forget the technical and spiritual support provided by Roberto. I was really blessed by joining SSRG at the same time Roberto did. Although I started collaboration with Sebastiano later, his contributions enriched the technical aspects of my dissertation, and every discussion with him added something to me. Mohamed is the kind of person that you can rely on – he is simply my brother.

I would like to thank my advisors and colleagues in the VT-MENA program, yet another great environment where you can always make achievements as a group. I owe Prof. Sedki Riad a lot for his endless, sincere guidance and advice, and also for all his efforts to make VT-MENA a successful story. I am also grateful to Prof. Mustafa Nainay and Prof. Hicham Elmongui for their continuous help and support.

I would like to thank my dear parents for being the main reason I am what I am. They worked hard in the past to raise me and prepare the way for my success, and then they suffered, but never complained, when they missed me for three years. May I be always the righteous son they wish. I am also grateful to everyone in my family, especially my brothers and sister, Ayman, Khaled, and Fatma, and my parents-in-law, for their continuous love and support.

My deepest love and gratitude is to my dearest wife, Mennatalla, who believed in me, supported me, and surrounded me and my lovely kids, Laila and Rokaia, with truthful love. May I always find the coolness of my eyes with her and with our kids.

Last but not least, I am blessed with being in Blacksburg and among its amazing community for more than three years. I am grateful to everyone whom I met here and everyone who gave me any sort of help, support, and advice. Special thanks to my dear friend, Mohamed Magdy and his family. I cannot find words to express my gratefulness for everything they made for me and my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

At the beginning of the new century, computer manufacturers faced difficulties in increasing the clock speed of processors because of hitting the overheating wall. Given that, multicore architectures have been introduced as an alternative to exploit the increasing number of transistors that can fit into the same space (according to Moore's low). In multicore architectures, increasing performance is achieved by adding more cores rather than increasing the processor's frequency.

Given this shift towards multicore machines, software developers were enforced to develop *concurrent programs* in which they have to synchronize the blocks of code that access the same regions of the shared memory (i.e., *critical sections*) simultaneously. Programmers used to protect their critical sections leveraging locks. On the hardware level, implementing efficient locks motivated the addition of more complex special (i.e., *atomic*) instructions, such as *compare-and-swap* (CAS). On the software level, synchronization using locking is very challenging. On one hand, coarse-grained locking, in which the shared objects are synchronized using a single global lock, is easy to program. However, it minimizes parallelism (concurrency) and prevents the full exploitation of the multicore computing resources. As a result, performance in coarse-grained locking is hardly better than executing critical sections sequentially. On the other hand, fine-grained locking, in which the programmer uses locks only when necessary, allows more concurrency but it is error-prone and more likely to suffer from issues like *race conditions* and *deadlocks*. To efficiently use fine-grained locking, the programmer has to ensure that those locks are *i)* sufficient to satisfy the required application correctness guarantees (e.g., consistency and isolation) and progress guarantees (e.g., deadlock freedom and fairness), and *ii)* optimized to allow as much concurrency as possible.

With the growing adoption of multicore processors, the design of efficient data structures that allow concurrent accesses without sacrificing true parallelism becomes more critical than before. In the last decade, different designs of the concurrent version of well-known data structures (e.g., lists, queues, hash tables, trees) have been proposed [59, 54, 46, 79, 29, 93, 13, 21, 22, 5, 8, 34, 15, 35, 63, 84]. They can be classified as lock-based (mostly fine-grained) designs and non-blocking (e.g., wait-free [56], obstruction-free [61]) designs. The former solutions are easier to design than non-blocking algorithms, but their performance could suffer from: *i)* the blocking nature of their operations; and *ii)* the possibility of delaying and/or stalling the lock holders (e.g., due to adverse operating system's scheduling). On the other hand, non-blocking algorithms use the atomic primitives (e.g., CAS operations) in a more efficient way in order to provide higher progress guarantees (e.g., wait-freedom [56]), which are otherwise prevented in the lock-based approaches [43].

1.1 Transactional Memory

In the last century, the adoption of a certain fine lock granularity has been an obligation for programmers. Similarly, programmers were responsible for selecting either lock-based or non-blocking designs for concurrent data structures. Along with that, Transactional Memory [58] (TM) has emerged as an appealing concurrency control methodology that shifts the burden of synchronization from the programmer to an underlying framework. With TM, programmers organize reads and writes to shared memory in "atomic blocks", which are guaranteed to satisfy atomicity, consistency, and isolation properties. Two transactions conflict if they access the same object and one access is a write. If two transactions conflict, one of them is aborted (to guarantee consistency), undoing all its prior changes. When a transaction commits, it permanently publishes its writes on shared memory. This way, other transactions (or at least successful ones) will not see its intermediate states, which guarantees atomicity and isolation. TM has been proposed in pure software [24, 25, 98, 31, 90, 37, 72, 86, 89], pure hardware [58, 7, 44, 82], and hybrid [28, 91, 27] approaches.

TM was introduced as a synchronization alternative to the classical lock-based approaches. The main goal of TM is to solve the tradeoff between performance an programmability. It provides high level of programmability, like coarse-grained locking, and it performs closer to the highly concurrent fine-grained locking applications. The underlying TM framework encapsulates all of the concurrency control low-level mechanisms and allows the programmer to write large and complex concurrent applications with possibly high correctness and progress guarantees.

Transactional memory is increasingly gaining traction: Intel has released a C++ compiler with STM support [64]; IBM [16] and Intel [65] have released commodity hardware processors with transactional memory support; GCC has released language extensions to support STM [99]. Having standard APIs for TM (like the recently released GCC API's) adds another advantage to be exploited in the coming TM research, which is transparency. Programmers can evaluate different TM algorithms/approaches and build different *what-if* scenarios using the same API's/benchmarks and with minimum programming overheads. This advantage is hard to achieve in the traditional lock-based approaches as they are usually application-dependent.

1.1.1 Software Transactional Memory

Inspired by database transactions, STM manages an atomic block by storing its memory accesses in local read-sets and write-sets. Read-sets (or write-sets) are local logs which store any memory location read (or written) within the transaction, and are used to validate the transaction at any time of its execution. To achieve consistency, a validation mechanism is used (either eagerly or lazily) to detect conflicting transactions (i.e., read-write or write-write conflicts). Writing to the shared memory is protected by locking the modified memory blocks until the transaction finishes its commit routine.

One of the most performance-critical decisions of an STM algorithm is when to write to shared memory. On one hand, eager writes (i.e., before commit) obligates a transaction to undo the writes in case of abort. The main problem of early updates is that writes of doomed transactions are visible to transactional and/or non-transactional code. On the other hand, lazy writes (i.e., during commit) are typically kept in a local redo logs to be published at commit, which solves the previous problem. Reads in this case are more complex, because, readers must scan their redo logs for the not-yet-committed writes, increasing the STM overhead. Another performance-critical design decision is the granularity of locking that STM algorithms use during commit. Commit-time locking can be extremely coarse-grained as in TML [24] and NOrec [25], compacted using bloom filters [12] as in RingSTM [98], or fine-grained using ownership records as in TL2 [31]. In addition, some algorithms replace the commit-time locking with an eager approach where locks are acquired at encounter time (e.g., TinySTM [90]). In general, fine-grained locking decreases the probability of unnecessary serialization of non-conflicting executions with an additional locking cost and more complex implementation, further affecting STM performance and scalability.

STM algorithms vary in properties such as progress guarantees, publication and privatization safety [77, 97], support for nesting [83, 100], interaction with non-transactional memory access, safe execution of exception handlers, irrevocable operations, system calls, and I/O operations. However, improving the performance and scalability are still the most important challenges in making STM a competitive alternative to traditional lock-based synchronization approaches, especially on emerging multi/many core architectures which offer capabilities for significantly increasing application concurrency.

We classified the main overheads that affect STM performance into three categories. First, the overhead of meta-data handling and validation/commit routines. Second, the locking mechanisms used in STM algorithms. Finally, the false conflicts raised from the intensive speculation.

The first overhead is the handling of meta-data. An STM transaction can be seen as being composed of a set of operations that must execute sequentially – i.e., the transaction's *critical path*. Reducing any STM operation's overhead on the critical path, without violating correctness properties, will reduce the critical path's execution time, and thereby significantly enhance STM's overall performance. These operations include *meta-data logging*, *locking*, *validation*, *commit*, and *abort*. Importantly, these parameters interfere with each other. Therefore, reducing the negative effect of one parameter (e.g., validation) may increase the negative effect of another (i.e., commit), resulting in an overall degradation in performance for some workloads.

Another distinguished overhead that affects the performance of STM is the locking mechanism itself. Most of the current STM frameworks, such as DEUCE in Java [67] and RSTM in C/C++ [73, 1], use spin locks in their STM algorithms to protect memory reads and writes. The evaluation of some recent locking mechanisms, like flat combining [55] (FC) and remote core locking [71] (RCL), showed better performance than spin locking. The basic idea of these mechanisms is letting threads spin on a *local* variable rather than a *shared* lock. This local variable is modified either by a nominated thread (in FC) or by a dedicated server (in RCL). Despite the efficiency of those approaches, they have never been used before in the TM context.

The last overhead in our study is the overhead of false conflicts, which are the conflicts that occur on memory when there is no semantic conflict and there is no need to abort the transaction. By nature, Transactional Memory is prone to false conflicts because it proposes a generic application-independent synchronization mechanism. Being unaware of the application logic may result in redundant speculation and unnecessary false conflicts. The problem of false conflicts is more evident in the context of data structures, which we cover with more details in Section 3.

1.1.2 Hardware Transactional Memory

TM was firstly proposed in hardware rather than in software, as a modification of the cache coherence protocols [58]. This way, HTM avoids the overhead incurred by the intensive speculation in STM. However, HTM did not attract computer architects for a while because it adds significant complications on the the design of the multicore architectures. The lack of commercial processors with HTM support enforced the research efforts to rely on simulations [44] to evaluate the proposed HTM solutions.

Recently, an important milestone in TM has been achieved by supporting HTM in both IBM's and Intel's commodity processors [16, 65]. HTM does not suffer from the same overheads of STM, and thus it potentially solves the TM's performance and scalability issue. However, the current HTM proposals come with two other serious limitations:

First, the common design principle in these releases is the *best-effort execution.* There is no guarantee that HTM transactions will eventually commit, independently of the surrounding circumstances. As an example, in Intel's Haswell processor with its TSX extensions, an HTM transaction may fail because of reasons other than conflicting with other transactions. One (important) of them is what we call *capacity failure*, which means that transaction's footprint (basically its reads and writes) does not fit the L1 cache size, which is private to each physical core and is used by TSX as buffer for logging transactional reads and writes. Other reasons include page faults, system calls invocations, and interrupts.

The architects' decision of releasing best-effort HTM architectures supports the direction of designing hybrid TM systems. This means that HTM transactions need a fall-back software-base path. Researcher, even before the releases of Intel's and IBM's processors, have recently enriched the literature with various proposals for the fall-back paths to either adapted locking mechanisms [4, 19] or STM mechanisms [91, 27, 75]. As another direction, HTM has been proposed to specifically enhance the design of concurrent and transactional data structures [9, 10, 101].

Second, Both Intel's [65] and IBM's [16] architectures provide very limited set of APIs to the programmer. As an example, intel TSX APIs allows identifying the transaction boundaries without giving the programmer the ability to execute part of it non-transactionally. IBM's Power8 APIs allows non-transactional code to be executed inside transactions using the new *suspend* instruction. However, recent work [95] showed a significant overhead of using *suspend* instruction intensively. It is worth noting that the earlier STM frameworks, such as RSTM [1] support non-transactional reads/writes within transactions.

Having such a limited API reduces the space of innovation in the execution of the *pure-HTM* transactions (before falling back to the software path). It also lets HTM inherit the problem of having intensive *false conflicts*, especially in data structures which are the core of our study, because HTM "blindly" monitors all the memory reads and writes within a transaction, even if there is no "semantic" need for monitoring them.

1.2 From Concurrent to Transactional Data Structures

With the advent of TM in multicore architectures, programmers are expected to move towards the "transactional" programming model, where transactions represent the basic building blocks for synchronization. Since concurrent data structures are key actors in the parallel programming model, and they are complex as well given the semantics that rules their operations, migrating them to the new (simpler) transactional programming model should help programmers (especially those non-expert) in developing multi-threaded applications that encompass data structures. In this dissertation, we make the first step towards identifying and overcoming the challenges of designing transactional, rather than concurrent, data structures. Specifically, we discuss three main issues that should be addressed when designing transactional data structures, such that composability, integration with generic transactions, and modeling; and along with that we introduce innovative solutions to address them.

1.2.1 Composability

One of the main limitations of concurrent data structures is that they do not compose. For example, as shown in Figure 1, all transactions access a shared set *set1*. Although *set1* could be implemented using an efficient highly concurrent data structure, such as lazy [54] linked-list and Contention-Friendly Tree [22], atomically inserting two elements in *set1* is difficult: if the `add` method internally uses locks, issues like managing the dependency between operations executed in the same transaction, and the deadlock that may occur because of the chain of lock acquisitions, may arise. Similarly, composing non-blocking operations is challenging because of the need to atomically modify different places in the data structure using only basic primitives, such as a CAS operation. Lack of composability is a serious limitation of current data structure designs, especially for legacy systems, as it makes their integration with third-party software difficult.

Algorithm 1 An example of using Atomic blocks to execute two data structure operations.

```
1: Set set1 = new(Set)

2: procedure METHOD1($x$)
3:     TM-BEGIN
4:     set1.add($x$)
5:     set1.add($y$)
6:     TM-END
7: end procedure
```

Although the research has reached an advanced point in developing concurrent data structures, transactional (i.e., composable) data structures have not reached this point yet. There are two practical approaches, to the best of our knowledge, that enable transactional accesses on a data structure: *1)* The first approach is Transactional Memory (TM) [58] which natively allows composability as it speculates every memory access inside an *atomic* block; *2)* The second approach is Transactional Boosting [57] (TB), which protects the transactional access to a concurrent data structure with a set of *semantic* locks, eagerly acquired before executing the operation on the concurrent data structure. Both TM and TB have serious limitations when used for designing transactional data structures. Those limitations originate from the same reason: they are both generic, and they do not consider the specific characteristics of concurrent data structures, which instead are heavily investigated in literature. For example, TM considers every step in the operation, as low-level memory reads/writes, which increases the number of false conflicts. On the other hand, TB uses the underlying concurrent data structure as a black-box, which prevents any further customization, and may nullify the internal optimizations of the concurrent data structure itself due to the eagerly acquired *semantic* locks.

To overcome these downsides, we present an optimistic methodology for boosting concurrent collections, called *Optimistic Transactional Boosting* (OTB). Unlike TM and TB, OTB only provides guidelines to design transactional data structures, and it leaves all the development details to the data structure designer, thus enabling the possibility of adding further (data structure-specific) optimizations. OTB is clearly less programmable than TM and TB, but it has the potential to provide better performance and scalability, especially when applied to complex data structures, like the case of skip-lists and balanced trees. Also, since we propose encapsulating the highly efficient OTB-Based transactional data structures in a software library similar to `java.concurrent.util` library, the programming efforts would be only made once by the data structure designer, and application level high programmability will be still preserved.

1.2.2 Integration

Since "transactions" are the most appealing candidates to replace locks as synchronization primitives, transactional data structures become no longer standalone components. For example, in Algorithm 2, all transactions access a shared set (*set1*), and two shared integers (*n1* and *n2*), which hold respectively the number of successful and unsuccessful `add` operations on *set1*. In *method1*, both the set operation (Line 6) and the traditional memory accesses (Lines 7 and 9) have to be executed atomically as one transactional block, without breaking the transaction's consistency and isolation.

To execute such an atomic block, the previous proposals use a pure TM-based implementation of *set1*, which lets the TM framework instrument the whole transaction. Conversely, in this dissertation we introduce techniques to extend TM frameworks so that they can use a more efficient implementation of *set1* (typically using the OTB methodology), rather than the fully-instrumented TM implementation, and at the same time allows for an atomic execution of this type of "mixed" transactions (i.e., embracing efficient data structure accesses and classical memory accesses), without suffering from false conflicts.

Algorithm 2 An example of using Atomic blocks to execute both data structure operations and memory reads/writes.

```
 1: Set set1 = new(Set)
 2: integer n1 = 0
 3: integer n2 = 0

 4: procedure METHOD1($x$)
 5:     TM-BEGIN
 6:     if set1.add($x$) == true then
 7:         TM-WRITE(n1, TM-READ(n1) + 1)
 8:     else
 9:         TM-WRITE(n2, TM-READ(n2) + 1)
10:     TM-END
11: end procedure
```

Integrating OTB with TM frameworks gives a global perspective to our work: providing generic transactional frameworks that execute both memory-based operations and data structure operations with *i)* as simple API as TM, and *ii)* as efficient "semantic" operations as OTB. In that sense, enhancing the performance of the overall TM framework (including the traditional memory-based operations) becomes also essential for preserving the performance gains of OTB. For that reason, in this dissertation we analyze the limitations of both STM and HTM transactions (discussed in Sections 1.1.1 and 1.1.2 respectively) and provide novel solutions for addressing them.

Specifically, regarding STM, we analyze the parameters that affect the critical path of STM transaction execution, and summarize the earlier attempts in the literature to reduce their effects. Then, we present two novel STM algorithm: *remote transaction commit* (RTC) and *remote invalidation* (RInval), which alleviate the overhead on the critical path. Also, in both algorithms, we address the issue of spin locking during transactions' commit phases. To the best of our knowledge, RTC and RInval are the first STM algorithms that make use of more enhanced locking mechanisms.

On the HTM side, we propose injecting the semantics of OTB data structures into the software fallback path of HTM transactions. We applied this idea on three HTM algorithms: the default HTM with global locking in the fallback path; HyNOrec [26, 88]; and NOrecRH [74], and we show how the new algorithms perform significantly better than their traditional "non-semantic" versions.

1.2.3 Modeling

The design of a data structure has its own challenges that depend on its semantics and implementation constraints. That is why, historically, proving the correctness of most concurrent data structures followed an ad-hoc approach. This lack of generality contributed to make the task of assessing their correctness very challenging. Recently, we observed an initial step towards accomplishing the goal of having a general model for proving the correctness of concurrent data structures, which is the *single writer multiple readers* model (we name it *SWMR* hereafter) presented by Lev-Ari et. al. in [70]. The *SWMR* model focuses on two safety properties (roughly summarized here): *validity*, which guarantees that no "unexpected" behaviors (e.g., access to an invalid address or a division by zero) can occur in all the steps of a concurrent execution; and *regularity*, an extension of the classical regularity model on registers [68] that guarantees that each read-only operation is consistent (i.e., linearized) with all the write operations. The appealing advantages of the *SWMR* model are that: *i)* it allows the programmer to use general and well-defined terms to prove the *validity* and *regularity* of any data structure fitting the *SWMR* model; and *ii)* it gives a formal way to prove *linearizability* [60] by relying on *regularity*.

Despite the strengths of the *SWMR* model, the set of data structures that can actually benefit from it does not include most of the recent highly optimized and practical concurrent [54, 46, 22] data structures which, in addition, allow concurrent writes. In addition, SWMR model assumes concurrent data structures and does not include transactional (or composable) data structures, like the OTB-based ones.

We build two models that extend *SWMR*: the first one covers the set of *OTB-based* concurrent data structures; and the second one extends the former to support the *composition* using OTB. We do so by leveraging two general observations about OTB-based data structures. First, the *traversal* phases do not execute any write and thus they can be considered as *internal read-only* operations. Second, the *commit* phases can atomically run with *Single Lock Atomicity* [78] semantics, rather than executing sequentially. Relying on those observations, we replace the assumption of having a *single writer* in *SWMR* with "more practical" assumptions. The main advantage of our models is that they allow the programmer to use well-defined terms, like *validity* and *regularity*, for proving the correctness of this practical set of data structures, instead of the ad-hoc proof techniques used so far.

1.3 Summary of Dissertation Contributions

In this dissertation we present innovative solutions to design, develop, and model transactional data structures.

To address *composability*, we design *Optimistic Transactional Boosting* (OTB) [48], an optimistic methodology for converting concurrent data structures into transactional ones. The main challenge of optimistic boosting is to ensure comparable (or better in some cases) performance to the highly concurrent data structures, while providing transactional support. Our approach follows the general optimistic rule of deferring operations to transaction commit. Precisely, transactional operations do not directly modify the shared data structure at encounter-time. Instead, they populate their changes into local logs during their execution. This way, OTB combines the benefits of concurrent data structures (i.e., un-monitored traversals), TB (i.e., semantic validation), and TM (i.e., optimistic conflict detection).

We apply OTB on both list-based data structures and balanced trees. We first developed a transactional version of linked-list-based set, skip-list-based set, and skip-list-based priority queue [47]. Then, we move to more complex data structure and introduce TxCF-Tree [51], the transactionally boosted version of CF-Tree [22], namely one of the most efficient and recent concurrent balanced tree. In addition to providing a "transactional" rather than a "concurrent" behavior of TxCF-Tree, we show how OTB can be leveraged to address one of the main issues of balanced trees, which is the overhead of the *structural operations* (i.e., rotations and physical deletion). TxCF-tree addresses this issue by introducing novel techniques to minimize the interference of such *structural operations* on the critical path of the *semantic operations* (e.g., queries, logical removals, and insertions).

Next, we show how to integrate transactionally boosted data structures with the current TM frameworks [49]. More specifically, we show how to implement OTB data structures in a standard way that can integrate with STM frameworks. Our frameworks are designed so that the integration between semantic operations and memory accesses is completely transparent to the programmer. Programmer just write the classical atomic block (delimited by `TM-BEGIN` and TM-END) enclosing semantic operations, like that shown in Algorithms 1 and 2.

As a test case, we show how to modify both DEUCE [67] and RSTM [1] frameworks to allow this integration while maintaining the consistency and programmability of the framework. Using the proposed integration, OTB transactional data structures are supposed to work in the context of generic transactions. That is why the proposed integration gains the benefits of both STM and boosting. On one hand, it uses OTB data structures with their minimal false conflicts and optimal data structure-specific design, which increases their performance. On the other hand, it keeps the same simple STM interface, which increases programmability. To the best of our knowledge, this linking between transactional data structures and STM algorithms has not been investigated in literature before.

We also show how to exploit the recently released HTM-enabled processors when designing transactional data structures. As we mentioned before, all HTM algorithms have two paths of execution: a *fast* HTM path, and a *slow* software fall-back path. The fall-back path is clearly slow because it either uses a global lock that serializes all the ongoing transactions, or uses a relatively slow STM algorithm such as NOrec [26, 88]. The fast-path, on the other hand, is not always as fast as it should be because it may repeatedly fail due to the HTM limitations mentioned before. Our methodology boosts the capabilities of the HTM algorithms by *i)* injecting the *efficient* semantic operations into the slow-path, and *ii)* using an adaptation mechanism to decide for each transaction which is the most effective path to start with. This approach makes significant performance improvements when data structure operations are dominating because, by relying on the first point, the slow-path will perform *faster* due to the exploitation of the efficient semantic versions of those operations, and, by relying on the second point, the fast-path can be bypassed when it is *slower* than the slow-path (e.g., when it repeatedly fails). To evaluate our methodology, we extended RSTM to support both the original *non-semantic-based* HTM algorithms and our *semantic-based* HTM algorithms. Our experiments show a better performance than the HTM algorithms that access the data structure without exploiting its semantics.

We then aim at enhancing the overall performance of the proposed transactional frameworks. To do so, we present *Remote Transaction Commit* (RTC) [52], a mechanism for executing commit phases of STM transactions. Specifically, RTC dedicates server cores to execute transactional commit phases on behalf of application threads. This approach has two major benefits. First, it decreases the overhead of spin locking during commit, in terms of cache misses, blocking of lock holders, and CAS operations. Second, it enables exploiting the server cores for more optimizations, such as running two independent commit phases in two different servers. As argued by [30], hardware overheads such as CAS operations and cache misses are critical bottlenecks to scalability in multicore infrastructures. Although STM algorithms proposed in the literature cover a wide range of locking granularity alternatives, they do not focus on the locking mechanism (which is one of our focuses in this thesis).

Next, we present *Remote Invalidation* (RInval) [50], an STM algorithm which applies the same idea of RTC in invalidation-based STM algorithms [40]. This way, RInval optimizes locking overhead because all spin locks and CAS operations are replaced with optimized cache-aligned communication. Additionally, like other invalidation-based STM algorithms, RInval's commit servers are responsible for invalidating the in-flight conflicting transactions. A direct result of this approach is reducing the execution-time complexity of validation to be linear instead of quadratic (as a function of the read-st size), because it avoids incremental validation after each memory read. We also introduce an enhanced version of RInval, in which the commit routine only publishes write-sets, while invalidation is delegated to other dedicated servers, running in parallel, thereby improving performance.

Finally, we address the issue of modeling concurrent and transactional data structures by providing an answer to the following question: can we define a model for assessing their correctness so that current (and maybe future) practical implementations can rely on that? Seeking such an answer, we present a set of models that cover different data structures designs [53]: *i) Single Writer Commit (SWC)*, which allows the traversal phases of *update* operations to run concurrently (as internal *read-only* operations), and enforces their *commit* phases to be atomically executed under the *single lock atomicity* (SLA) semantics [78]. The main challenge addressed by this model is that the *traversal* phases, unlike the *SWMR* model, are not standalone *read-only* operations but are tightly related with their corresponding *commit* phases; ii) Composable Single Writer Commit (C-SWC): which is an extension of *SWC* that allows the composition of the operations by grouping the *traversal* phases (respectively *commit* phases) in a single transactional *traversal* phase (respectively *commit* phase). The main challenge here is that operations in the same transaction have to be able to exchange the information about their *traversal* and *commit* phases.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows. We overview past work in Chapter 2. We describe OTB methodology in Chapter 3, and we show how we applied it on list-based data structures and balanced trees in Chapters 4 and 5, respectively. Then, in Chapters 6 and 7, we show how we integrate OTB with STM frameworks and HTM algorithms, respectively. Chapters 8 and 9 describe our RTC and RInval STM algorithms, respectively. Then, we describe SWC and C-SWC models in Chapter 10. Finally, Chapter 11 concludes the dissertation and discusses potential future work.

Chapter 2

Background

2.1 STM algorithms

Since the first proposal of Software Transactional Memory implementation (i.e., Transactional Locking [94]), many STM algorithms with different designs have been introduced in literature. In this section, we overview past STM algorithms that are most relevant to our work, and contrast them with our algorithms. Basically, RTC is a validation-based algorithm which extends NOrec [25], and RInval is an invalidation-based algorithm which extends InvalSTM [40]. Also, different works propose NOrec as an efficient fallback path to HTM transactions. Thus, we describe in details these two algorithms in Section 2.1.1 and Section 2.1.2 respectively. Then, in Section 2.1.3, we briefly discuss other STM algorithms and how they interleave with our algorithms.

2.1.1 NOrec

NOrec [25] is a lazy STM algorithm which uses minimal meta-data. Only one global timestamped lock is acquired at commit time to avoid write-after-write hazards. When a transaction T_i attempts to commit, it tries to atomically increment a global timestamp (using a CAS operation), and keeps spinning until the timestamp is successfully incremented. If the *timestamp* is odd, this means that some transaction is executing its commit phase. In this case, the read-set has to be validated before retrying the CAS operation. After the lock is acquired (i.e., CAS succeeds), the write-set is published on the shared memory, and then the lock is released.

Validation in NOrec is value-based. After each read, if the transaction finds that the timestamp has been changed, it validates that the values in its read-set match the current values in the memory.

Using single global lock and value-based validation avoids the need to use *ownership records – or orecs* (the name NOrec stands for *no ownership records or no orecs*). Orecs were used in many STM algorithms [31, 90] as the meta-data associated with each memory block to detect and resolve conflicts between transactions. Some other STM algorithms, such as RingSTM [98], use bloom filters instead of orecs. NOrec, however, does not use neither orecs nor bloom filters, and it limits the meta-data used to be only the global sequence lock. Having this minimum meta-data allows low read/write overhead, easy integration with HTM, and fast single-thread execution. Additionally, value-based validation reduces false conflicts because the exact memory values are validated instead of the orecs.

One of the issues in NOrec is that it uses an incremental validation mechanism. In incremental validation, the entire read-set has to be validated after reading any new memory location. Thus, the overhead of read-validation is a quadratic function of the read-set size. NOrec alleviates this overhead by checking the global lock before validation. If the global lock is not changed, validation is skipped. However, the worst case complexity of the validation process in NOrec remains quadratic. Another issue is that commit phases have to be executed serially.

2.1.2 InvalSTM

The invalidation approach has been previously investigated in [45, 62, 40]. Among these approaches, Gottschlich *et. al* proposed commit-time invalidation, (or InvalSTM) [40], an invalidation algorithm that completely replaces version-based validation without violating opacity [42].

The basic idea of InvalSTM is to let the committing transaction invalidate all active transactions that conflict with it before it executes the commit routine. More complex implementation involves the contention manager deciding if the conflicting transactions should abort, or the committing transaction itself should wait and/or abort, according to how many transactions will be doomed if the committing transaction proceeds, and what are the sizes of their read-sets and write-sets.

Like NOrec, committing a transaction T_i starts with atomically incrementing a global timestamp. The difference her is that after writing in memory, T_i invalidates any conflicting transaction by setting their *status* flag as *invalidated.* Conflict detection is done by comparing the write bloom filters [12] of the committing transaction with the read bloom filters of all in-flight transactions. Bloom filters are used because they are accessed in constant time, independent of the read-set size. However, they increase the probability of false conflicts because bloom filters are only compact bit-wise representations of the memory.

When T_i attempts to read a new memory location, it takes a snapshot of the timestamp, reads the location, and then validates that timestamp does not change while it reads. Then, T_i checks the *status* flag to test if it has been invalidated by another transaction in an earlier step. This flag is only changed by the commit executor.

The invalidation procedure replaces incremental validation, which is used in NOrec [25]. Thus, the overhead of read-validation becomes a linear function of the read-set size instead of a quadratic function. This reduction in validation time enhances the performance, especially for memory-intensive workloads. It is worth noting that both incremental validation and commit-time invalidation have been shown to guarantee the same correctness property, which is opacity [42].

One of the main disadvantages of commit-time invalidation is that it burdens the commit routine with the mechanism of invalidation. In a number of cases, this overhead may offset the performance gain due to reduced validation time. Moreover, InvalSTM uses a conservative coarse-grained locking mechanism, which of course makes its implementation easier, but at the expense of reduced commit concurrency (i.e., only one commit routine is executed at a time). The coarse-grained locking mechanism increases the potential of commit "over validation", because the commit executor will block all other transactions that attempt to read or commit. Other committing transactions will therefore be blocked, spinning on the global lock and waiting until they acquire it. Transactions that attempt to read will also be blocked because they cannot perform validation while another transaction is executing its commit routine (to guarantee opacity).

2.1.3 Other STM algorithms

RingSTM [98] introduced the idea of detecting conflicts using bloom filters [12]. Each thread locally keeps two bloom filters, which represent the thread's read-set and write-set. All writing transactions first join a shared ring data structure with its local bloom filters. Readers validate a new read against the bloom filters of writing transactions, which join the ring after the transaction starts. Although both RingSTM and RTC use bloom filters, there is a difference in the use of those bloom filters. RingSTM uses bloom filters to validate read-sets and synchronize writers, which increases false conflicts according to bloom filter sizes.

TL2 [31] is also an appealing STM algorithm which uses ownership records. However, this extremely fine grained speculation is not compatible with RTC's idea of remote execution because it will require dedicating more servers.

DSTM [62] is an example of partial invalidation which eagerly detects and resolves write-write conflicts. STM^2 [66] proposes executing validation in parallel with the main flow of transactions. However, it does not decrease the time complexity of incremental validation (like InvalSTM). Moreover, it does not guarantee opacity and needs a sand-boxing mechanism to be consistent [23].

Another alternative to an STM algorithm for supporting STM-style atomic sections is global locking, which simply replaces an atomic block with a coarse grained lock (for example, using MCS [76]). Although such an STM solution is suitable for applications which are sequential by nature, it is too conservative for most workloads, hampering scalability. STM runtimes like RSTM [73, 1] use such a solution to calculate the single thread overhead of other algorithms, and to be used in special cases or in adaptive STM systems.

2.2 Best-effort HTM Limitations

Since Intel [65] and IBM [16] announced their first processors with HTM support, the research of the TM community started deviating from the traditional direction of enhancing the performance and scalability of STM algorithms to finding the best solution to overcome the *best-effort* nature of HTM transactions.

Minimizing the side-effects of *best-effort* HTM architectures requires finding the best fall-back software path (either to global locking [18, 33] or to lightweight STM algorithms [74, 26, 88]), which can be implemented in Haswell using the *Restricted Transactional Memory* (RTM) APIs of the TSX extensions. To implement such an efficient fall-back path, it is important to understand the different sources of HTM failures. In this section we briefly discuss the sources of failure in the TSX extensions of Intel's Haswell processor, since we use Intel architectures in our frameworks. However, the general concepts discussed here apply to IBM's power8 as well.

The first type of failure, called *capacity failure*, is directly inherited from the capacity limitation on the underlying L1 cache (recall that TSX extensions are built on top of the cache coherence protocol). This type of failure enables the *_xabort_capacity* flag. Proposals to annul the effect of *capacity failures* include: tuning the number of retries in HTM before falling back to the software path [33]; and proposing new hardware instruction, like IBM's *suspend* operation [16], to reduce the signature of HTM transactions.

The second type of HTM failures is called *conflict failure* (detected via the $\_xabort\_conflict$ flag). This failure is the result of any conflict happened either due to the access on application data, or accessing the meta-data shared between the HTM path and the fall-back paths (like the global lock). Most of the research in HTM focuses on reducing conflicts on the meta-data [26, 88, 18, 74] because conflicts on the application data are considered as natural and unavoidable conflicts.

Finally, HTM transactions can abort by external sources (e.g., page faults, system calls, timer interrupts). Those kinds of aborts raise the *_xabort_other* flag.

2.3 Remote Core Locking

Remote Core Locking (RCL) [71] is a recent mutual exclusion mechanism based on the idea of executing lock-based critical sections in remote threads. Applying the same idea in

STM is appealing, because it makes use of the increasing number of cores in current multi-core architectures, and at the same time, allows more complex applications than lock-based approaches.

The main idea of RCL is to dedicate some cores to execute critical sections. If a thread reaches a critical section, it will send a request to a server thread using a cache-aligned requests array. Unlike STM, both the number of locks and the logic of the critical sections vary according to applications. Thus, RCL client's request must include more information than RTC, like the address of the lock associated with the critical section, the address of the function that encapsulates the client's critical section, and the variables referenced or updated inside the critical section. Re-engineering, which in this case means replacing critical sections with remote procedure calls, is also required and made off-line using a refactoring mechanism [39].

RCL outperforms traditional locking algorithms like MCS [76] and Flat Combining [55] in legacy applications with long critical sections. This improvement is due to three main enhancements: reducing cache misses on spin locks, reducing time-consuming CAS operations, and ensuring that servers that are executing critical sections are not blocked by the scheduler.

On the other hand, RCL has some limitations. Handling generic lock-based applications, with the possibility of nested locks and conditional locking, puts extra obligations on servers. RCL must ensure livelock freedom in these cases, which complicates its mechanism and requires thread management. Also, legacy applications must be re-engineered so that critical sections can be executed as remote procedures. This problem specifically cannot exist in STM because the main goal of STM is to make concurrency control transparent from programmers. As we will show later, RTC does not suffer from these limitations, retaining all the benefits of RCL.

An earlier similar idea is Flat Combining [55], which dynamically elects one client to temporarily take the role of server, instead of dedicating servicing threads. However, simply replacing spin locks in STM algorithms with RCL locks (or Flat Combining locks) is not the best choice because of two reasons. First, unlike lock-based applications, STM is a complete framework that is totally responsible for concurrency control, which allows greater innovation in the role of servers. Specifically, in STM, servers have more information about the read-set and write-set of each transaction. This information cannot be exploited (for improving performance) if we just use RCL as is. Second, most STM algorithms use sequence locks (not just spin locks) by adding versions to each lock. These versions are used in many algorithms to validate that transactions always see a consistent snapshot of the memory. Sequence locks cannot be directly converted to RCL locks while maintaining the same STM properties unless it is modified by mechanisms like RTC. In conclusion, RTC can be viewed as an extension of these earlier lock-based attempts to maintain all their benefits and adopt them for use inside STM frameworks.

2.4 Transactional Boosting

Herlihy and Koskinen's transactional boosting methodology [57](TB) enables transactions to run on top of a concurrent data structure object by defining a set of commutativity rules. Two operations are said to be commutative for a data structure object if their execution in either order will transition the (shared) object to the same state and return the same result. To support transactional operations, transactional boosting relies on the so called *semantic synchronization layer*. This layer is composed of: (i) abstract locks, which are used on top of the linearized data structure object to prevent non-commutative operations from running concurrently; and (ii) a semantic undo log, which is used to save operations to be rolled back in case of abort. This way, both *synchronization* and *recovery* of the transactions are guaranteed. Each operation acquires the necessary abstract locks to guarantee synchronization (abstract locks are released at the end of the transaction, either it commits or aborts). Saving the inverse operations in an undo-log guarantees transactions' recovery.

We argue that this protocol is pessimistic: locks are acquired at encounter-time, and writes are eagerly published. The semantic layer of locking boosts a concurrent object to be transactional. It also uses a simple interface, which wraps concurrent data structures as black boxes.

Pessimistic semantic locking has the following downsides:

- Although abstract locking in TB prevents boosted operations from conflict with each other, it does not natively cope with STM validations, and may suffer from doomed transactions if data structures are accessed outside its interface, either by transactional or non-transactional memory accesses). Additionally, adding eager abstract locks to guarantee opacity is costly, especially for lock-free operations which can be converted to be blocking operations, as we will show in details in Chapter 3.

- TB has limitations when integrated with STM frameworks. Even though it saves the overhead of monitoring unnecessary memory locations, its semantic two-phase locking mechanism is different from the mechanism of STM frameworks (which usually use read-sets and write-sets to monitor shared memory accesses). Using an optimistic boosting approach, which is similar to the mechanism of STM frameworks, allows easier and more efficient integration.

- TB assumes: *a)* well defined commutativity rules on the data structure operations and *b)* the existence of an inverse operation for each operation. If both these rules cannot be defined for a data structure, then the boosting methodology cannot be applied.

- Decoupling the boosting layer from the underlying concurrent data structure may result in losing the possibility of providing data structure-specific optimizations. In general, decoupling is a trade-off. Although decoupling the underlying data structures as black-boxes means that there is no need re-engineer their algorithms, it does not optimize these algorithms for the new transactional specifications, especially when the re-engineering can be easily achieved.

Chapter 3

Optimistic Transactional Boosting

3.1 Motivation

The increasing ubiquity of multi-core processors motivated the development of data structures that can exploit the hardware parallelism of those processors. The current widely used concurrent collections of elements (e.g., Linked-List, Skip-List, Tree) are well optimized for high performance and ensure isolation of atomic operations, but they do not *compose* (as mentioned early in Section 1.2.1). For example, Java's *Concurrent Collections* yield high performance for concurrent accesses, but require programmer-defined *synchronized* blocks for demarcating transactions. Such blocks are trivially implemented using coarse-grain locks that significantly limit concurrency. This is a significant limitation from a programmability standpoint, especially for legacy systems as they are increasingly migrated onto multicore hardware (for high performance) and must seamlessly integrate with third-party libraries.

Software transactional memory (STM) can be used to implement transactional data structures, which makes them composable – a significant benefit. However, monitoring all of the memory locations accessed by a transaction while executing data structure operations is a significant overhead. As a result, STM-based transactional collections perform inferior to their optimized, concurrent (i.e., non-transactional) counterparts.

One of the main overheads in STM-based transactional data structures is that monitoring all of the memory locations accessed by a transaction results in *false conflicts*. For example, if two transactions are trying to insert two different items into a linked-list, these two insertions are usually commutative (i.e., they are supposed to be executed concurrently without breaking their correctness). However, STM may not be able to detect this commutativity, and can raise a false conflict, aborting one of them.

Figure 3.1 shows an example of false conflicts in a linked-list. If a transaction $t1$ tries to insert 55 in the shown list, it has to put all of the traversed nodes (gray nodes in the figure) in its local read-set. Assume now that another transaction $t2$ concurrently inserts 4 (by modifying the link of 2 to point to 4 instead of 5). In this case $t1$ will abort because the nodes 2 and 5 are in its read-set. This is a false conflict because inserting 55 should not be affected by the concurrent insertion of 4. In some cases, like long linked-lists, these false conflicts dominate any other overheads in the system. Importantly, most of the efficient concurrent (non-transactional) linked-lists, such as lazy and lock-free linked-list [59], do not suffer from this false conflict.

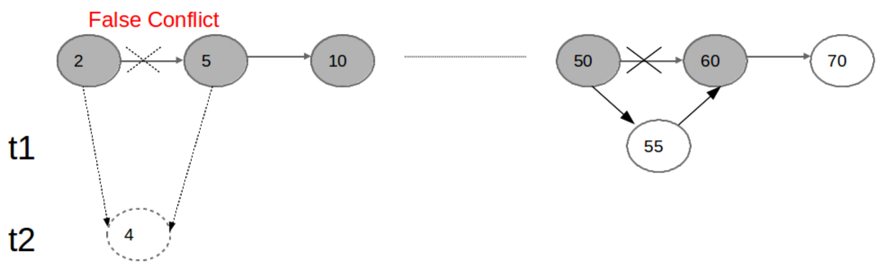


Figure 3.1: An example of false conflicts in a linked-list.

Recent works in literature propose different ways to implement transactional data structures other than the traditional use of STM algorithms. One direction is to adapt STM algorithms to allow the programmer to control the semantic of the data structures. Examples of trials in this direction include elastic transaction [38], open nesting [85], and early release [62]. Another direction is to use STM as support to design libraries of data structures. Examples in this direction include transactional collection classes [20], transactional predication [14], and speculation friendly red-black tree [21].

Another appealing alternative is Herlihy and Koskinen's *Transactional Boosting* (TB) [57] methodology, which converts concurrent data structures to transactional ones by providing a *semantic layer* on top of existing concurrent objects. However, as we showed in Section 2.4, TB has downsides that limit its applicability. Those downsides are mainly raised because in TB *i)* the abstract lock acquisition and modifications in memory are eager, and *ii)* the technique uses the underlying concurrent data structure as a black box. To overcome these downsides, in this chapter we present an optimistic methodology for boosting concurrent collections, called *Optimistic Transactional Boosting* (OTB). OTB allows data structure-specific optimizations, an easier integration with TM frameworks (as we will detail in Chapters 6 and 7), and less restrictions than TB on the boosted operations.

3.2 Methodology

Optimistic transactional boosting (OTB) [48] is a methodology to boost optimistic data structures (also known as lazy [54]) to be transactional. A common feature that can be identified in all lazy data structures is that they have an *unmonitored traversal* step, in which the object's nodes are not kept locked until the operation ends. To guarantee consistency, this unmonitored traversal is followed by a validation step before the last step that physically modifies the shared data structure. OTB modifies the design of these lazy data structures to support transactions. Basically, the OTB methodology can be summarized in three main guidelines.

(G1) *Each data structure operation is divided into three steps.*

- *Traversal*. This step scans the data structure, and computes the operation's results (i.e., its postcondition) and what it depends on (i.e., its precondition). This requires us to define (in each transaction), what we call *semantic read-set* and *semantic write-set*, which store these information (*semantic write-sets* can also be called *semantic redo-logs*).
- *Validation*. This step checks the validity of the preconditions. Specifically, the entities stored in the semantic read-set are validated to ensure that operations are consistent.
- *Commit*. This step performs the modifications to the shared data structure. Unlike concurrent data structures, this step is not done at the end of each operation. Instead, it is deferred to the transaction's commit time. All information needed for performing this step are maintained in the semantic write-sets during the first step (i.e., traversal). To publish the write-sets, a classical (semantic) two-phase locking is used. This semantic (or abstract) locking prevents semantic conflicts at commit.

(G2) *Data structure design is adapted to support opacity.* The correctness of transactional data structures does not only depend on the *linearization* of its operations (like concurrent data structures), but it also depends on the sequence of the operations executed in each transaction. Data structure design has to be adapted to guarantee this *serialization* part. OTB provides the following guidelines, which exploit the local read-sets and write-sets to guarantee opacity [42] (In Section 4.1.3, we prove that those guidelines are sufficient to guarantee opacity.), the same consistency level of most STM algorithms [25, 31, 89, 62]:

(G2.1) Each operation scans the local write-set first, before accessing the shared object. This is important to include the effect of the earlier (not yet published) operations in the same transaction.

(G2.2) The read-set is re-validated after each operation and during commit, to guarantee that each transaction always observes a consistent state of the system (even if it will eventually abort).

(G2.3) During commit, semantic locks of all operations are acquired before any physical modification on the shared data structure.

(G2.4) Operations are applied during the commit phase in the same order as they appeared in the transaction and, in case the outcome of an operation influences the subsequent operations recorded in the write-set, they are updated accordingly.

(G2.5) All operations have to be validated, even if the original (concurrent) operation does not make any validation (like `contains` operation in set). The goal of validation in these cases is to ensure that the same operation's result occurs at commit.

(G3) *Data structure design is adapted for more optimizations.* Each data structure can be further optimized according to its own semantic and implementation. For example, in set, if an item is added and then deleted in the same transaction, both operations eliminate each other and can be completed without physically modifying the shared data structure.

Unlike the first two guidelines, which are general for any lazy data structure, the third guideline varies from one data structure to another. It gives a hint to the developers that the data structures now are no longer used as black boxes, and further optimizations can be applied. It is important to note that the generality of the first two guidelines does not mean that they can be applied "blindly" without being aware of the data structure's semantics. OTB, like TB, performs better than the naive STM-based data structures only because it exploits semantics. However, we believe that OTB's guidelines make a clear separation between the general outline that can be applied on any lazy data structure (like validation, in *G2.2*, and commit, in *G2.4*, even if the validation/commit mechanisms themselves vary from one data structure to another) and the specific optimizations that are completely dependent on the data structures implementation. Evidences of such a claim are in Chapters 4 and 5 where we deploy these guidelines to develop a transactional lazy set, priority queue, and tree.

3.3 Analyzing OTB

Using OTB to design transactional data structures is clearly better than using TM. This is because, unlike the classical meaning of read-sets and write-sets in STM (and also HTM, if we consider the L1 cache as an internal read-set and write-set), not all memory reads and writes are saved in the semantic read-sets and write-sets. Instead, only those reads and writes that affect linearization of the object and consistency of the transaction are saved. This avoids the *false conflicts* discussed in Section 3.1.

To compare OTB with TB (we also call it *pessimistic boosting*), Figure 3.2 shows the execution flow of: concurrent (lazy) data structures, TB, and OTB. Concurrent (non-transactional) data structures yield high performance because they traverse the data structure without instrumentation, and they only acquire locks (or use CAS operations in case

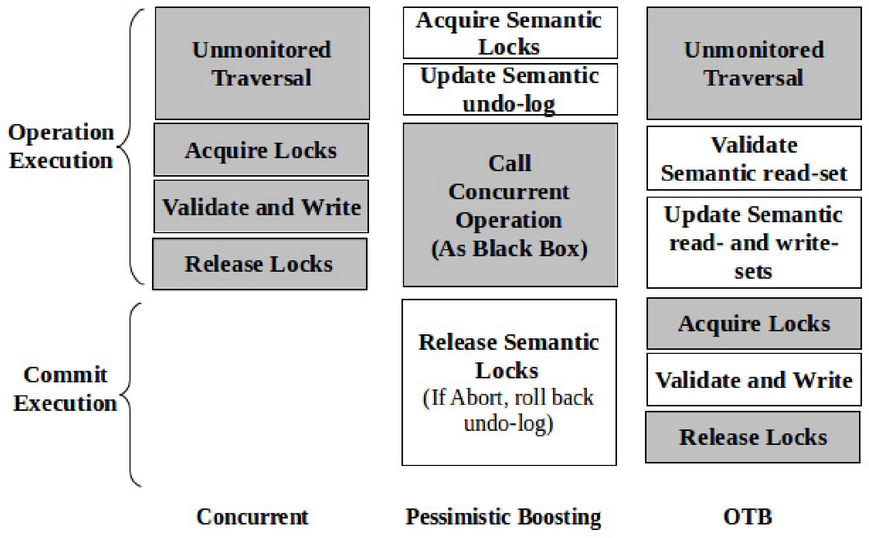


Figure 3.2: Execution flow of: concurrent (lock-based or lock-free) data structures; TB (Pessimistic Boosting); OTB.

of lock-free objects) at late phases. To add transactional capabilities, pessimistic boosting acquires semantic locks eagerly, and saves the inverse operations in an undo-log (to rollback the transaction in case of abort). Then, it uses the underlying concurrent data structure as a black box without any modifications. (In both TB and OTB, dark blocks in Figure 3.2 are the same as the concurrent versions, while white blocks are added/modified.) At commit time, the only task to be accomplished is the release of semantic locks, because operations have already been executed eagerly.

In contrast to pessimistic boosting, OTB acquires semantic locks lazily, and uses the underlying data structure as a white box. Similar to concurrent data structures, OTB traverses objects without instrumentation. However, it differs from them in three aspects: *i)* lock acquisition and actual writes are shifted to commit time; *ii)* the validation procedure is modified to satisfy the new transactional requirements; and *iii)* the necessary information is saved in local semantic read-sets and write-sets.

Thus, OTB gains the following benefits over pessimistic boosting. First, it does not require well defined commutativity rules or inverse operations. Second, integration with STM frameworks is easy, as OTB uses the same phases of validation and commit (with the same meaning as in STM). Third, it uses highly concurrent collections as white boxes to design new transactional versions of each concurrent (non-transactional) data structure. This allows greater optimizations according to the new transactional features, with minimal re-engineering overhead.

3.4 The Optimistic Semantic Synchronization trend

A set of recent methodologies leverage the same idea of OTB: dividing the transaction execution into phases and optimistically executing some of them without any instrumentation (also called *unmonitored* phases).

We use the term *Optimistic Semantic Synchronization* (OSS) to represent this set of methodologies. We used the word *optimistic* because all of these solutions share a fundamental optimism by dividing transactions into phases. In this section, we overview some of those approaches.

Consistency Oblivious Programming (COP) [3, 9, 11] splits the operations into the same three phases as OTB (but under different names). We observe two main differences between COP and OTB. First, COP is introduced mainly to design concurrent data structures and it does not natively provide composability unless changes are made at the hardware level [11]. Second, COP does not use locks at commit. Instead, it enforces atomicity and isolation by executing both the *validation* and *commit* phases using STM [3] or HTM [9] transactions.

Partitioned Transactions (ParT) [101] also uses the same trend of splitting the operations into a *traversal* (called *planning*) phase and a *commit* (called *update*) phase, but it gives more general guidelines than OTB. Specifically, ParT does not restrict the planning phase to be a traversal of a data structure and it allows this phase to be any generic block of code. Also, ParT does not obligate the planning phase to be necessarily unmonitored, as in OTB and COP. Instead, it allows both the planning and update phases to be transactions.

Transactional Predication [14] applies a similar methodology to the aforementioned approaches. However, it solves the specific problem of boosting concurrent sets and maps to be transactional.

We consider this line of research as a confirmation of OTB's effectiveness. We also keep all those approaches into consideration when we discuss the other challenges towards designing and modeling transactional data structures. In fact, although we focus on OTB (as the main contribution of this thesis) when discussing the integration of transactional data structures with generic transactions (Chapters 6-7) and modeling transactional data structures (Chapter 10), the solutions proposed also apply to the other methodologies just overviewed.

3.5 Summary

In this chapter we presented *Optimistic Transactional Boosting*, an optimistic methodology to convert optimistic concurrent data structures to transactional ones. Transactions use semantic read-set and write-set to locally save the operations of boosted objects, and defer modifying the shared objects to the commit phase. Optimistic boosting can be easily integrated with current STM systems, while keeping the same correctness and progress guarantees. Instead of using concurrent data structures as black boxes, optimistic boosting is used to implement new transactional versions of concurrent collections that allow for effective low-level optimizations given the new transactional characteristics.

Chapter 4

Transactional List-Based Data Structures

In this chapter we present two types of optimistically boosted data structures: set and priority queue. These were specifically chosen as they represent two different categories:

- *Commutable Objects.* In set, operations are commutative at the level of keys themselves. In other words, two operations are semantically commutative if they access two different keys in the set.
- *Non-commutable Objects.* Priority queue operations are commutative at the level of the whole object. This means that, even if two operations access two different items in the queue, they cannot execute in parallel. In fact, any `removeMin` operation is non-commutative with another `removeMin` operation as well as any `add` operation of items that are smaller than the removed minimum.

Despite this difference, the design and implementation of optimistically boosted versions of both the data structures follow the same basic principles illustrated in section 3.2, with slight modifications to cope with the different levels of commutativity.

4.1 Set

Set is a collection of ordered items, which has three basic operations: `add`, `remove`, and `contains`, with the familiar meanings [59]. No duplicate items are allowed (thus, `add` returns false if the item is already present in the structure). All operations on different items of the *set* are commutative – i.e., two operations `add(x)` and `add(y)` are commutative if $x \neq y$. Moreover, two `contains` operations on the same item are commutative as well. Such a high degree of commutativity between operations enables fine-grained semantic synchronization.

Lazy linked-list [54] is an efficient implementation of concurrent (non transactional) *set*. For write operations, the list is traversed without any locking until the involved nodes are locked. If those nodes are still valid after locking, the write takes place and then the nodes are unlocked. A `marked` flag is added to each node for splitting the deletion phase into two steps: the logical deletion phase, which simply sets the flag to indicate that the node has been deleted, and the physical deletion phase, which changes the references to skip the deleted node. This flag prevents traversing a chain of deleted nodes and returning an incorrect result. It is important to note that the `contains` operation in the lazy linked-list is wait-free and is not blocked by any other operation.

Lazy skip-list is, in general, more efficient than linked-list as it takes logarithmic time to traverse the *set*. In skip-list, each node is linked to multiple lists (i.e., levels), starting from the list at the bottom level (which contains all the items), up to a random level. Therefore, `add` and `remove` operations lock an array of *pred* and *curr* node pairs (in a unified ascending order of levels to avoid deadlock), instead of locking one pair of nodes as in linked-list. For `add` operation, each node is enriched with a $fullyLinked$ flag to logically add it to the *set* after all levels have been successfully linked. Skip-list is also more suited than linked-list in scenarios where the overhead of rolling back (compared to execution) is dominating. In fact, for a linked-list (and especially a long linked-list), even if aborts are rare, their effect includes re-traversing the whole list again, in a linear time, to retry the operation. In a skip-list, the cost of re-traversal is lower (typically in a logarithmic time), which minimizes the overhead of the aborts.

The implementation of the transactional boosted (i.e., TB [57]) version of the *set* is straightforward and does not change if the *set* implementation itself changes. In fact, it uses the underlying concurrent lazy linked-list (or skip-list) to execute the *set* operations. If the transaction aborts, a successful `add` operation is rolled back by calling the `remove` operation on the same item, and vice versa (more details are in [57]).

Despite the significant improvement in the traversal cost and abort overhead, the implementation of OTB skip-list and OTB linked-list are very similar. With the purpose of making the presentation clear, we focus on the linked-list implementation, and we highlight the main differences with respect to the skip-list implementation when necessary.

4.1.1 Non-Optimized OTB-Set

Following the first two guidelines (i.e., *G1* and *G2*) as mentioned in Section 3.2, in this section we show how to boost the lazy *set* to design a transactional *set* without any specific optimization related to the details of its implementation. According to *G1*, we divide OTB-Set operations into three steps. The *Traversal* step is used to reach the involved nodes, without any addition to the semantic read-set. The *Validation* step is used to guarantee the consistency of the transaction and the linearization of the list. We define two different validation procedures: one is named *post-validation*, which is called after each operation, and the other is named *commit-time-validation*, which is called at commit time and after acquiring the semantic locks. The Commit step, which modifies the shared list, is deferred to transaction's commit. Following *G2*, we show how the usage of lazy updates, semantic locking, and post-validation guarantees opacity.

Similar to the lazy linked-list, each operation in OTB-Set involves two nodes at commit time: *pred*, which is the largest item less than the searched item, and *curr*, which is the searched item itself or the smallest item larger than the searched item[1]. To log the information about these nodes, with the purpose of using them at commit time, we adopt the same concept of read-set and write-set as used in lazy STM algorithm (e.g., [25, 31]), but at the semantic level. In particular, each read-set or write-set entry contains the two involved nodes in the operation and the type of the operation. In addition, the write-set entry contains also the new value to be added in case of a successful `add` operation.

The only difference in skip-list is that the read-set and write-set entries contain an array of *pred* and *curr* pairs, instead of a single pair. This is because the searched object can be in more than one level of the skip-list.

Algorithm 3 OTB Linked-list: `add`, `remove`, and `contains` operations.

```
1:  procedure OPERATION($x$)
                                        ▷ Step 1: search local write-sets
2:      if $x \in$ write-set then
3:          ret = write-set.get-ret(op,x)
4:          if op is add or remove then
5:              write-set.append(op,x)
6:          return ret
                                        ▷ Step 2: Traversal
7:      $pred = head$ and $curr = head$.next
8:      while $curr$.item $< x$ do
9:          $pred = curr$
10:         $curr = curr.next$
                                        ▷ Step 3: Save reads and writes
11:     rse = new ReadSetEntry($pred$,$curr$,op)
12:     read-set.add(rse)
13:     if op is add or remove then
14:         wse = new WriteSetEntry($pred$,$curr$,op,$x$)
15:         write-set.add(wse)
                                        ▷ Step 4: Post Validation
16:     if ¬ post-validate(read-set) then
17:         ABORT
18:     else if Successful operation then
19:         return $true$
20:     else
21:         return $false$
22: end procedure
```

Algorithm 3 shows the pseudo code of the linked-list operations. We can isolate the following four parts of each operation.

- *Local writes check* (lines 2-6). Since writes are buffered and deferred to the commit phase, this step guarantees consistency of further reads and writes. Each operation on an item x checks the last operation in the write-set on the same item x and returns the corresponding result. For example, if a transaction previously executed a successful `add` operation of item x, then further additions of x performed by the same transaction must be unsuccessful and return false. In addition, if the new operation is a writing (i.e., `add/remove`) operation, it should be appended to the corresponding write-set entry (line 5). If there is no previous (local) operation on x in the write-set, then the operation starts traversing the shared linked-list as shown in the next step.

[1]Sentinel nodes are added as the head and tail of the list to handle special cases.

- *Traversal* (lines 7-10). This step is the same as in the lazy linked-list. It saves the overhead of all unnecessary monitoring during traversal that, otherwise, would be incurred with a native STM algorithm for managing concurrency.

- *Logging the reads and writes* (lines 12-15). At this point, the transaction records the accessed nodes, that are semantically relevant to the *set*, into its local read-set and write-set. All operations must add the appropriate read-set entry, while `add/remove` operations modify also the write-set (line 15). It is worth to note that having no entries in the write-set for `contains` operation means that it does not need to acquire locks during the commit phase. This way, although the `contains` operation is no longer wait-free, like its concurrent lazy version (because it may fail during the commit-time-validation), it still performs efficiently due to the absence of the semantic locks acquisition. We recall that, rather than OTB, TB has to acquire semantic locks even for the `contains` operation to maintain consistency and opacity.

- *Post-Validation* (lines 16-21). At the end of the traversal step, the involved nodes are stored in local variables (i.e., *pred* and *curr*). At this point, according to point *G2.2* and to preserve opacity [42], the read-set is post-validated to ensure that the transaction does not observe an inconsistent snapshot. The same post-validation mechanism is used at memory-level by STM algorithms such as NOrec [25]. More details about post-validation are discussed later in Algorithm 4.

As mentioned before, there is a difference between linked-list and skip-list regarding the `add` operation. In fact, in the skip-list the new node has to be linked to multiple levels, thus there could be a time window where the new node is only linked to some (and not all) levels. To handle this case in our OTB-Set, any concurrent operation waits until the $fullyLinked$ flag becomes true, and then it proceeds.

Algorithm 4 shows the post-validation step. The validation of each read-set entry is similar to the one in lazy linked-list: both *pred* and *curr* should not be deleted, and *pred* should still link to *curr* (lines 6-8). According to *G2.5* of OTB guidelines, `contains` operation has to perform the same validation as `add` and `remove`, although it is not needed in the concurrent version. This is because any modification made by other transactions after invoking the `contains` operation and before committing the transaction may invalidate the returned value of the operation, making the transaction's execution semantically incorrect.

To enforce isolation, a transaction ensures that its accessed nodes are not locked by another writing transaction during validation. This is achieved by implementing locks as *sequence locks* (i.e., locks with version numbers). Before the validation, a transaction records the versions of the locks if they are not acquired. If some are already locked by another transaction, the validation fails. (lines 2-5). After the validation, the transaction ensures that the actual locks' versions match the previously recorded versions (lines 9-12).

Algorithm 5 shows the commit step of OTB-Set. Read-only transactions have nothing to do during commit (line 2), because of the incremental validation during the execution of

Algorithm 4 OTB Linked-list: validation.

```
1: procedure VALIDATE(read-set)
2:     for all entries in read-sets do
3:         get snapshot of involved locks
4:         if one involved lock is locked then
5:             return false
6:     for all entries in read-sets do
7:         if $pred$.deleted or curr.deleted or $pred$.next $\neq$ $curr$ then
8:             return false
9:     for all entries in read-sets do
10:        check snapshot of involved locks
11:        if version of one involved lock is changed then
12:            return false
13:    return true
14: end procedure
```

the transaction. For write transactions, according to point *G2.3*, the appropriate locks are first acquired using CAS operations (lines 4-6). Like the original lazy linked-list, any `add` operation only needs to lock *pred*, while `remove` operations lock both *pred* and *curr*. As described in [54], this is enough for preserving the correctness of the write operations. To avoid deadlock, any failure during the lock acquisition implies aborting and retrying the transaction (releasing all previously acquired locks).

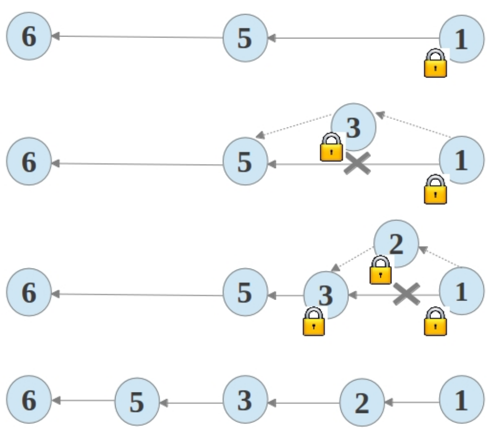
After the semantic lock acquisition, the validation is called, in the same way as in Algorithm 4, to ensure that the read-set is still consistent (line 7). If the commit-time-validation fails, then the transaction aborts.

Algorithm 5 OTB Linked-list: commit.

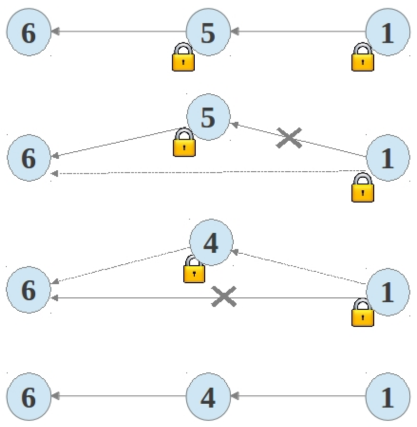
```
1: procedure COMMIT
2:     if write-set.isEmpty then
3:         return
4:     for all entries in write-sets do
5:         if CAS Locking $pred$ (or $curr$ if remove) failed then
6:             ABORT
7:     if ¬ commit-validate(read-set) then
8:         ABORT
9:     sort write-set descending on items
10:    for all entries in write-sets do
11:        $curr$ = $pred$.next
12:        while $curr$.item < $x$ do
13:            $pred$ = $curr$
14:            $curr$ = $curr$.next
15:    if operation = add then
16:        $n$ = new Node(item)
17:        $n$.locked = true
18:        $n$.next = $curr$
19:        $pred$.next = $n$
20:        for all entries in write-sets do
21:            if entry.$pred$ = $pred$ then
22:                entry.$pred$ = $n$
23:    else                                          ▷ remove
24:        $curr$.deleted = true
25:        $pred$.next = $curr$.next
26:        for all entries in write-sets do
27:            if entry.$pred$ = $curr$ then
28:                entry.$pred$ = $pred$
29:            else if entry.$curr$ = $curr$ then
30:                entry.$curr$ = $curr$.next
31:    for all entries in write-sets do
32:        unlock $pred$ (and $curr$ if remove)
33: end procedure
```

The next step of the commit procedure is to publish writes on the shared linked-list, and then release the acquired locks. This step is not straightforward because each node may be involved in more than one operation of the same transaction. In this case, the saved *pred* and *curr* of these operations may change according to which operation commits first.

For example, in Figure 4.1(a), both 2 and 3 are inserted between the nodes 1 and 5 in the same transaction. During commit, if node 2 is inserted before node 3, it should be the new predecessor of node 3, but the write-set still records node 1 as the predecessor of node 3. In



(a) Two `add` operations (2 and 3).



(b) `add(4)` and `remove(5)`.

Figure 4.1: Executing more operations that involve the same node in the same transaction.

OTB guidelines, *G2.4* solves this issue. When node 2 is inserted, the operation scans the write-set again to find any other operation that has node 1 as its *pred* and replaces it with node 2. The same technique is used in the case of removal (Figure 4.1(b)). When node 5 is removed, any write-set entry that has node 5 as its *curr* replaces it with node 6, and any write-set entry that has node 5 as its *pred* replaces it with node 1. Lines 20-22 and 26-30 illustrate these cases.

It is clear that the inserted nodes have to be locked until the whole commit procedure is finished. Then they are unlocked along with the other *pred* and *curr* nodes (line 17). For example, in Figure 4.1(a), all nodes (1, 2, 3, 5) are locked and no transaction can access them until the commit terminates.

4.1.2 Optimized OTB-Set

One of the main advantages of OTB over TB is that it uses the underlying (lazy) data structure as a white-box, which allows more data structures-specific optimizations.

In general, decoupling the boosting layer from the underlying concurrent data structure is a trade-off. Although, on the one side, considering the underlying data structure as a black-box means that there is no need to re-engineer its implementation, on the other side, it does not allow to customize its implementation and thus to exploit the new transactional specification, especially when the re-engineering effort can be easily achieved. For this reason, as showed in the previous section, we decided to split the re-engineering efforts (required by OTB) into two steps: one general (concluded in OTB guidelines *G1* and *G2*); and one more specific per data structure (concluded *G3*). We believe this division makes the re-engineering task easier and, at the same time, it allows specific optimizations for further enhancing the performance.

In this section, we show optimizations for our OTB-Set, leveraging the fact that it treats the underlying lazy linked-list as a white-box and, therefore, it can be adapted as needed.

Unsuccessful `add` and `remove`.

The `add` and `remove` operations are not necessarily considered as writing operations, because duplicated items are not allowed in the *set*. For example, if an `add` operation returns false, it means that the item to insert already exists in the *set*. The commit of such operation can be done by only checking that the item still exists in the *set*, which allows to treat unsuccessful `add` operations as successful `contains` operations. This way, the transaction does not acquire any lock for this operation at commit. The same idea can be applied on the unsuccessful `remove` operation which can be treated as an unsuccessful `contains` operation during commit.

Accordingly, in our OTB-Set, both `contains` and unsuccessful `add/remove` operations are considered as read operations (which add entries only to the semantic read-set and do not acquire any semantic locks during commit). Only successful `add` and `remove` operations are considered read/write operations (which add entries to both the read-set and the write-set and thus acquire semantic locks during commit).

In the lazy linked-list, the `add` and `remove` operations acquire locks on the *pred* and *curr* nodes even if the operations are unsuccessful. TB inherits this unnecessary lock acquisition because it uses the lazy linked-list as a black-box.

Eliminating Operations.

As shown in Algorithm 3, each operation starts with checking the local writes before traversing the shared list. During this step, for improving OTB performance, if a transaction adds an item x and then removes the same item x, or vice versa, we allow those operations to locally eliminate each other. This elimination is done by removing both entries from the write-set, which means that the two operations will not make any physical modification on the shared list. No entry in the read-set is locally eliminated because, this way, the commit time-validation can still be performed on those operations in order to preserve transaction's correctness.

In TB, due to the usage of the underlying lazy linked-list as a black-box, this scenario is handled by physically adding x to the shared *set*, and then physically removing it, introducing an unnecessary overhead.

Simpler Validation.

In the case of successful `contains` and unsuccessful `add` operations, we use a simpler validation than the original validation of the lazy linked-list. In these particular cases, the transaction only needs to check that *curr* is still not deleted, since that is sufficient to guarantee that the returned value is still valid (recall that if the node is deleted, it must first be logically marked as deleted, which will be detected during validation). This optimization prevents false invalidations, where conflicts on *pred* are not real semantic conflicts.

The validation in the skip-list is similarly optimized because we leverage the rule that all items have to appear in the lowest level of the skip-list. For successful *contains* and unsuccessful *add* operations, it is sufficient to validate that *curr* is not deleted, which ensures that the item is still in the *set*. We can also optimize unsuccessful *remove* and *contains* by only validating the *pred* and *curr* in the lowest level to make sure that the item is still not in the *set*, because if the item is inserted by another transaction, it must affect this level. For successful *add* and *remove* operations, all levels need to be validated to prevent conflicts.

Optimized Commit.

To ensure that the operations in Figure 4.1 are executed correctly, the write-set has to be re-scanned for each write operation (according to the OTB guideline *G2.4*), as we showed in Section 4.1.1. This overhead becomes significant if the write-set is relatively large. We optimize this routine and avoid the need of re-scanning the write-set by the following points. *(1)* The items are added/removed in descending order of their values, regardless of their order in the transaction execution. This guarantees that the *pred* of each write-set entry is always valid, non-deleted, and not touched by any previous operation in the transaction. *(2)* Operations resume traversal from the saved *pred* to the new *pred* and *curr* nodes. At this stage, the *pred* and *curr* nodes can only be changed because of some previous local operations. This is because the transaction already finished the lock acquisition and validation, which prevents any conflicting transaction from proceeding.

Using these two points, the issue in Figure 4.1(a) is solved without re-scanning the write-set. The first point enforces that node 3 is inserted first. Subsequently, according to the second point, when 2 is inserted, the transaction will resume its traversal from node 1 (which is guaranteed to be locked and non-deleted). Then, it will detect that node 3 is its new *succ*, and will correctly link node 2.

The removal case is shown in Figure 4.1(b), in which node 5 is removed and node 4 is inserted. Again, 5 must be removed as first (even if 4 is added earlier during the transaction execution), so that when 4 is added, it will correctly link to 6 and not to 5. Two subsequent `remove` operations follow the same procedure.

Skip-list uses the same procedure but at all levels. This is because each level is independent from the others, which means that the *preds* of the same node in two or more levels may be different. For this reason, the same procedure described above is repeated at each level, independently.

In the following algorithms, we show how to modify Algorithms 3, 4, and 5 to maintain these optimizations.

Algorithm 6 OTB Linked-list: Optimized `add`, `remove`, and `contains` operations.

```
 1: procedure OPERATION($x$)
                                        ▷ Step 1: search local write-sets
 2:     if $x \in$ write-set and write-set entry is add then
 3:         if operation = add then
 4:             return false
 5:         else if operation = contains then
 6:             return true
 7:         else                                              ▷ remove
 8:             delete write-set entry
 9:             return true
10:     else if $x \in$ write-set and write-set entry is remove then
11:         if operation = remove or operation = contains then
12:             return false
13:         else                                              ▷ add
14:             delete write-set entry
15:             return true
16:     ...
                                        ▷ Step 3: Save reads and writes
17:     read-set.add(new ReadSetEntry($pred$, $curr$, operation))
18:     if Successful add/remove then
19:         write-set.add(new WriteSetEntry($pred$, $curr$, operation, $x$))
20:     if Successful operation then
21:         return $true$
22:     else
23:         return $false$
24: end procedure
```

Algorithm 6 shows the modification to steps 1 and 3 of Algorithm 3 to achieve the first two optimizations. In step 1, eliminated operations are removed from the write-set, while still keeping them in the read-set (lines 8 and 14). In step 3, only successful `add` and `remove` operations are added to the write-set (line 19)

Algorithm 7 OTB Linked-list: optimized validation.

```
 1: procedure VALIDATE(read-set)
 2:     ...
 3:     for all entries in read-sets do
 4:         if successful contains or unsuccessful add then
 5:             if $curr$.deleted then
 6:                 return false
 7:         else
 8:             if $pred$.deleted or curr.deleted or $pred$.next $\neq curr$ then
 9:                 return false
10:             return true
11:     ...
12: end procedure
```

Algorithm 7 shows the optimized validation procedure. Lines 3-10 replace lines 6-8 in Algorithm 4.

Algorithm 8 OTB Linked-list: optimized commit.

```
1: procedure Commit
2:     ...
3:     sort write-set descending on items
4:     for all entries in write-sets do
5:         $curr = pred$.next
6:         while $curr$.item $< x$ do
7:             $pred = curr$
8:             $curr = curr$.next
9:     ...
10: end procedure
```

Algorithm 8 describes the modified commit procedure, which replace lines 20-22 and 26-30 of Algorithm 5). Line 3 applies the first guideline point in Section 4.1.2. Lines 5-8 apply the second guideline point.

4.1.3 Correctness of OTB-Set

In this section, we give more details about the correctness of OTB-Set. In Section 4.1.3 we assume, without loss of generality, transactions composed of only one OTB-Set operation and we prove that this operation is linearizable. Then, in Section 4.1.3, we prove that if the transaction contains more than one operation of a *non-optimized* OTB-Set, the implementation of OTB guidelines guarantees opacity [42]. Finally, Section 4.1.3 discusses the correctness of the *optimized* OTB-Set.

Linearization

Although OTB-Set does not use the lazy *set* as a black box, it uses similar mechanisms for traversing the *set* and validating the nodes accessed. For this reason, we can assess the correctness of our OTB-Set implementations by relying on the correctness of the lazy *set*. Again, we focus on a linked-list-based *set*, assuming that applying the same concepts to a skip-list-based *set* is straightforward.

- *Successful* `add`. As described in [54], a successful `add` operation in a lazy linked-list is linearized in the moment when *pred*.next is set. This is because, at this point, the `add` operation already *i)* acquired the lock on *pred*, *ii)* validated that *pred* and *curr* are not deleted and that *pred* still links to *curr*. This means that the operation is still valid and no other concurrent operation can interfere with it. In OTB-Set, the same linearization point is selected. We now assume that transactions consist of only one operation. This means that, until the moment of acquiring the lock on *pred*, both the lazy *set* and OTB-Set behave in a similar way, even though the lock acquisition itself is delayed to the commit phase in OTB-Set. Once the lock is acquired in OTB-Set (at commit), it uses the same validation as the lazy *set* which allows having the same linearization point.

- *Successful* `remove`. In the lazy linked-list, the successful `remove` operation is linearized when the entry is marked as *deleted*. We can safely select the same linearization point in OTB-Set because, although this point is shifted to the transaction's commit, OTB-Set still behaves similar to the lazy *set*, like the `add` operation.

- *Successful* `contains`. Here, the linearization points of lazy *set* and OTB-Set are different. In lazy *set*, the `contains` operation is wait-free and the successful `contains` operation is linearized immediately when the *deleted* flag of a matching entry is observed to be false. In OTB-Set, however, this point cannot be selected as the linearization point for two reasons. First, the linearization point should be selected somewhere in the commit phase, thus allowing the further extension of having more than one operation in the same transaction (that we will discuss in Section 4.1.3). Second, according to the point *G2.4* of the OTB guidelines, all operations, including the `contains` operation, should re-validate their results at commit, which means also that the operation may fail during the commit and therefore the transaction may abort. For this reason, we select the linearization point of the `contains` operations when those operations are successfully re-validated at commit time.

- *Unsuccessful* `contains`. Like successful `contains`, the unsuccessful `contains` operations are re-validated at commit time and we select this re-validation point as our linearization point. It is worth to note that the definition of the linearization point of the unsuccessful `contains` operations in the lazy *set* is not straightforward. In fact, in the lazy linked-list, the linearization point of a logically (but not physically) deleted node has to be before that any other transaction occasionally added a new node with the same item. However, this case in our OTB-Set is not relevant because we abort the transaction in this case (remember that we validate that both the *pred* and *curr* nodes are not logically deleted). Although this abort can be seen as a false conflict, we allow it for the sake of making a simple validation process, and assuming that such case is rare. The lazy *set* also suffers from a similar false conflict in the case of unsuccessful `remove`, and they also chose not to optimize this case for the sake of a simple and clear validation process.

- *Unsuccessful* `add/remove`. In the non-optimized version of the OTB-Set, described in Section 4.1.1, the linearization points of the unsuccessful `add/remove` operations are the same as in the lazy *set*. This is because, in this non-optimized version, the same locks (on the *pred* and *curr* nodes) are acquired and the same validation is used. In the optimized version of the OTB-Set, as described in Section 4.1.2, those linearization points change because we now consider the operations as read operations and we do not acquire any locks for them during commit. As we mentioned in Section 4.1.2, those operations are considered during commit as successful/unsuccessful `contains` operations, which motivates using the same linearization points as the `contains` operation in this case, rather than the linearization points mentioned in [54].

Opacity - Non-Optimized OTB-Set

Opacity was proposed in [42] to formally proof the correctness of TM implementations, and most STM algorithms are proven to guarantee opacity [25, 31, 89, 62]. Intuitively, as mentioned in [42], opacity is guaranteed if three requirements are captured: *i)* every committed transaction atomically appears in a single indivisible point in the history of the committed transactions, *ii)* live transactions do not see the intermediate results of any aborted transaction, *iii)* transactions are always consistent even if they will eventually abort. In this section and in the following section, we show that those three requirements are preserved in both the "non-optimized" and the "optimized" OTB-Set, respectively. The correctness of the non-optimized OTB-Set can be also used for any other lazy data structure as far as it follows the same two guidelines mentioned in Section 3.2 (i.e., *G1* and *G2*).

In the following, we borrow the same terminology used in [42]. However, for brevity, instead of having two points in the history for each operation (the *invocation* point and the *return* point), we will only show one point which reflects the *return* point. This is acceptable because any transaction is serial, meaning that it does not invoke a new operation until the previous operation returns its value, and the *invocation* point is not relevant to the execution of any other transaction.

- *Equivalence to a legal sequential history.* The first requirement for a history H to be opaque is that if we remove all non-committed transaction, the resulting sub-history H' is equivalent to a legal sequential history S that preserves the real-time order of the transactions in H'. In OTB-Set, H' preserves the real-time order because all operations are linearized during the commit phases of their transactions. For that reason, a committing transaction can be serialized in one point, right after the transaction successfully acquires its semantic locks. After this serialization point, if the transaction successfully validates its read-set, all conflicting transactions in H' will be serialized after it. If it fails in validation, it will simply abort.

  Precisely, we have five cases to cover for proving the legality of any sub-history H' of some committed OTB-Set transactions $T_1, T_2, ..., T_n$:

  1. Transaction are executed serially: which means that each transaction starts after the previous transaction commits. The real-time order in this case is natively preserved because after T_i commits, all its writes are immediately visible to the following transactions (threads are not caching any state of the objects).
  2. Concurrent transactions are independent (which means that they have no intersection in their read/write-sets or the intersection is only between read-sets). Natively, they can be serialized in any order. The history of each transaction as a standalone transaction is kept legal using the guidelines *G2.1* and *G2.4*. For example, in the following history:
     $H1 =< add(T_i, x, true), contains(T_i, x, true), remove(T_i, x, true), tryC_{T_i},$
     $C_{T_i} >$

G2.1 guarantees that both the `contains` and the `remove` operations cannot return an illegal value (which is false in this case) during the execution of the transaction, and *G2.4* guarantees that the `remove` operation will be executed correctly at commit (remember that we are now proving the non-optimized version of OTB-Set which means that operations are executed according to their invocation order and without any local elimination during commit).

3. The write-sets of two concurrent transactions, T_i and T_j, intersect. Clearly the commit phases of those transactions can never execute concurrently. Either one of them will fail in acquiring the semantic locks and thus will abort, or T_j will start its commit after T_i entirely finishes its commit and releases its locks, which allows serializing T_i before T_j.
4. The read-set of T_i intersects with the write-set of T_j and the read-set of T_j does not intersect with the write-set of T_i. In this case, T_i will either abort during the commit-time validation, or it will successfully finish its validation before T_j acquires the "conflicting" semantic locks. In the latter case, T_i can be safely serialized before T_j.
5. The read-set of T_i intersects with the write-set of T_j and the read-set of T_j intersects with the write-set of T_i. In this case, any scenario where both transactions concurrently commit is illegal. For example, in the following two histories[2]:
$H2 =< remove(T_k, x, true), remove(T_k, y, true), tryC_{T_k}, C_{T_k}, add(T_i, x, true),$
$contains(T_j, x, false), add(T_j, y, true), contains(T_i, y, false), tryC_{T_i}, C_{T_i},$
$tryC_{T_j}, C_{T_j} >$
$H3 =< remove(T_k, x, true), remove(T_k, y, true), tryC_{T_k}, C_{T_k}, add(T_i, x, true),$
$contains(T_j, x, true), add(T_j, y, true), contains(T_i, y, true), tryC_{T_i}, C_{T_i},$
$tryC_{T_j}, C_{T_j} >$
Both histories are illegal because the `contains` operations in T_i and T_j cannot return both false or both true[3]. A possible legal case is that the `contains` operation of T_i returns false and the one of T_j returns true (which allows T_i to be legally serialized before T_j).
Our validation process in Algorithm 4 prevents that all these illegal scenarios can happen. As we validate that the nodes in the read-set are both *unlocked* and *valid*. T_i and T_j cannot both successfully acquire the semantic locks and then successfully validate their read-sets before starting to write. At least one transaction will abort because some entries in its read-set is locked by the other transaction.

- *The effect of the aborted transactions.* Aborted transactions in OTB-Set have no effect on the live transactions. This is simply because transactions do not publish any writes until their commit phase. During commit, if a transaction successfully acquires the semantic locks and then it successfully validates its read-set, it cannot abort anymore. Accordingly, it is safe at this point to start writing on the shared *set*.

[2]We put the first two operations of T_k to enforce that x and y are both in the *set* before T_i and T_j start.
[3]This case is an example of producing a cyclic opacity graph which is mentioned in [42].

- *Consistency of live transactions.* Transactions which guarantee opacity should always observe a consistent state. This also includes the *live* transactions, which are the transactions that did not yet commit or abort. Theoretically, as mentioned in [42], we can transform any history which contains some live transactions to a complete history by either committing or aborting those live transactions. The challenge here is to prove that this completed history is still legal (which means that the operations executed so far inside the live transactions are legal). In OTB-Set we guarantee that live transactions always observe a consistent state by the *post-validation* procedure which validates, after each operation, that the entire read-set is still valid. Precisely, in a history H, an operation $< op(T_i, x, true/false) >$, can be implicitly extended to either $< op(T_i, x, true/false), validate(T_i, succeeded) >$ or $< op(T_i, x, true/false), validate(T_i, failed), A_{T_i} >$ according to whether its validation succeeds or fails*, which guarantees preserving the legality of H.*

Opacity - Optimized OTB-Set

In this section we show how each optimization discussed in Section 4.1.2 does not prevent transactions to guarantee opacity.

- *Unsuccessful `add/remove`.* OTB-Set validates the unsuccessful `add/remove` operations as `contains` operations. It can be easily shown that this does not break transactions' consistency. Although operations are semantically different, handling them in the same way at commit (at the memory level) does not break the semantics with any means, as far as the same result is validated during commit.

- *Eliminating Operations.* Elimination does not break consistency because operations are eliminated only from the write-sets. If the operations were also eliminated from the read-sets, opacity may be broken because another transaction may modify the *set* and then commit successfully before the commit of the former transaction, which violates the serialization points we mentioned in the previous section. For example, in the following history: $H4 =< add(T_i, x, true), remove(T_i, x, true), add(T_j, x, true), tryC(T_j), C(T_j), tryC(T_i),$ $C(T_i) >$ transaction T_i becomes illegal. This is because at the serialization point of T_i, when it commits, the `add` operation cannot return true because T_j already added x to the *set* and committed. In our OTB-Set implementation, T_i will detect during its commit that T_j added x because the eliminated operations are still in the read-set and they will be validated during commit, and thus triggering the abort of T_i.

- *Simpler Validation.* For the successful `contains` and the unsuccessful `add` operations, the *curr* node is detected to match the searched item x, and to be not *deleted*. During the commit phase of a transaction T_i, it is sufficient to check the *deleted* mark of the *curr* node. This is because any other transaction cannot execute any new writing operation on x before deleting the previous node, and deleting x is done first by logically mark the node as *deleted*. This means that, if T_i observes at commit that x is not logically deleted, then the operation is still valid. In this case, there is no need to validate the *pred* node.

- *Optimized Commit.* The correctness of this optimization is based on two facts. First, as write operations on the same item are eliminating each other, we cannot observe two entries of a transaction T_i's write-set, which add (or remove) the same item x. This means that all operations in the write-set are commutative (i.e., not semantically conflicting), and can be (semantically) executed in any order. Second, at memory level, it becomes also unnecessary to execute those write operations in the same order as their original order, because each operation does not change the *pred* of any subsequent operation (as they are sorted in a descending order). As the *pred* node is not changed, it is safe for any operation to start from that *pred* node to reach the new *pred* and *curr* nodes.

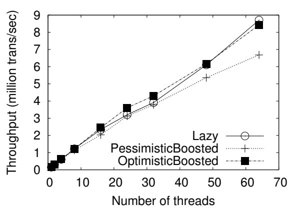
4.1.4 Evaluation

In this section we evaluate the performance of our OTB-Set's Java implementation equipped with the optimizations described in Section 4.1.2 (*OptimisticBoosted* in the plots). We compared it with lazy *set* [54] and TB *set* [57] (*PessimisticBoosted* in the plots). In order to conduct a fair comparison, the percentage of the writes in all of the experiments is the percentage of the successful ones, because an unsuccessful `add/remove` operation is considered as a read operation. Roughly speaking, in order to achieve that, the range of elements is made large enough to ensure that most `add` operations are successful. Also, each `remove` operation takes an item added by previous transactions as a parameter, such that it will probably succeed. In each experiment, the number of `add` and `remove` operations are kept equal to avoid significant fluctuations of the data structure size during the experiments.

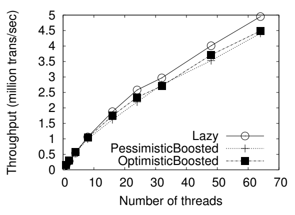
The experiments were conducted on a 64-core machine, which has four AMD Opteron (TM) Processors, each with 16 cores running at 1400 MHz, 32 GB of memory, and 16KB L1 data cache. Threads start execution with a warm up phase of 2 seconds, followed by an execution of 5 seconds, during which the throughput is measured. Each plotted data-point is the average of five runs.

We use transactional throughput as our key performance indicator. Although abort rate is another important parameter to measure and analyze, it is meaningless in our case. Both lazy *set* and TB *set* do not explicitly abort the transaction. However, there is an internal retry for each operation if validation fails. Additionally, TB aborts only if it fails to acquire the semantic locks, which is less frequent than validation failures in the OTB-Set. We recall that the lazy *set* is not capable to run transactions at all (i.e., it is a concurrent data structure, not transactional). We only show it as a rough upper bound for the OTB-Set and TB, but it actually does not support transactional operations.

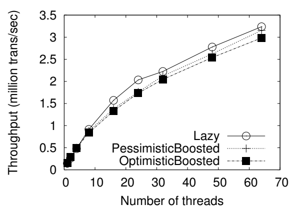
We first show the results for a linked-list implementation of the *set*. In this experiments, we used a linked-list with 512 nodes. In order to conduct a comprehensive evaluation of



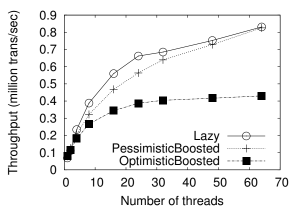
(a) LL 0%



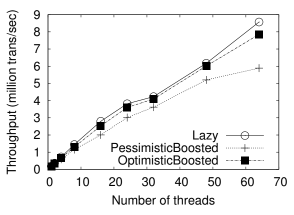
(b) LL 20%



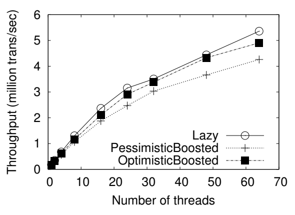
(c) LL 80%



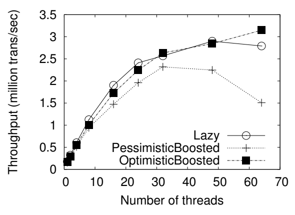
(d) LL 80% and 5 ops



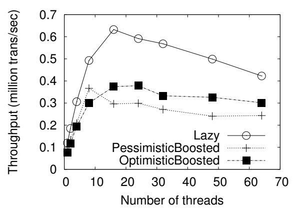
(e) SL 0%



(f) SL 20%



(g) SL 80%

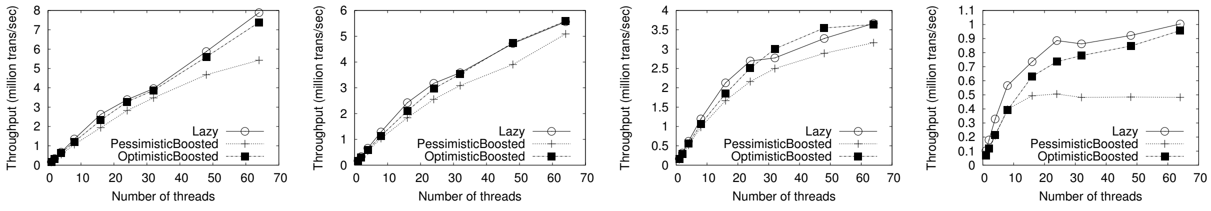


(h) SL 80% and 5 ops

Figure 4.2: Throughput of linked-list-based (LL) and skip-list-based (SL) set with 512 elements (labels indicate % write transactions). Four different workloads: read-only (0% writes), read-intensive (20% writes), write-intensive (80% writes), and high contention (80% writes and 5 operations per transaction).

OTB-Set's performance, in the first row of Figure 4.2 we show the results for four different linked-list workloads: read-only (0% writes and 1 operation per transaction), read-intensive (20% writes and 1 operation per transaction), write-intensive (80% writes and 1 operation per transaction), and high contention (80% writes and 5 operations per transaction). In both read-only and read-intensive workloads, OTB-Set performs closer to the (upper bound) performance of the lazy list than TB-Set. This is expected, because TB incurs locking overhead even for read operations. In contrast, OTB-Set, like lazy linked-list, does not acquire locks on read operations, although it still has a small overhead for validating the read-set. For the write-intensive workload, TB starts to be slightly better than OTB-Set, and the gap increases in high contention workloads. This is also expected, because contention becomes very high, which increases abort rate (recall that aborts have high overhead due to re-traversing the list in linear time). In these high/very high contention scenarios, the "pessimism" of TB pays off more than the "optimism" of OTB-Set. For example, in the high contention scenario, five operations are executed per transaction. In TB, each operation (pessimistically) locks its semantic items before executing each operation and then it keeps trying to execute the operation on the underlying (black-box) concurrent data structure. On the other hand, OTB suffers from aborting the whole transaction even if the last operation of the transaction fails.

In the second row of Figure 4.2, the same results are shown for the skip-list-based *set* of the same size (512 nodes). The results show that OTB-Set performs better in all cases, including the high contention case. This confirms that OTB-Set gains because of the reduced overhead of aborts. Although the semantic contention is almost the same (for a *set* with 512 nodes, contention is relatively high), using a skip-list instead of a linked-list supports OTB-Set more than TB. This is mainly because skip-list traverses less nodes of the set through the higher levels of the skip-list. Thus, even if the whole transaction aborts, re-executing skip-list operations is less costly than linked-list.



(a) SL 0% (b) SL 20% (c) SL 80% (d) SL 80% and 5 ops

Figure 4.3: Throughput of skip-list-based set with 64K elements (labels indicate % write transactions). Four different workloads: read-only (0% writes), read-intensive (20% writes), write-intensive (80% writes), and high-contention (80% writes and 5 operations per transaction).

The last set of experiments (Figure 4.3), shows the performance when the contention is significantly lower. We used a skip-list of size 64K and measured throughput for the same four workloads. The results show that in such cases, which however are still practical, OTB-Set is up to 2× better, even in write-intensive and high contention workloads. This is mainly because in the very low contention scenario, the TB's eager locking mechanism becomes ineffective and a more optimistic algorithm, such as OTB-Set, is preferable.

4.2 Priority Queue

Priority queue is a collection of totally ordered keys with duplicates allowed, and provides three APIs: `add(x)/-`, `min()/x`, and `removeMin()/x`.

In addition to the well-known heap implementation of priority queue, skip-list has also been proposed for implementing priority queue [59]. Although both implementations have the same logarithmic complexity, skip-list does not need periodic re-balancing, which is more suited for concurrent execution. Generally, cooperative operations such as re-balancing increase the possibility of conflict and degrade concurrency. Also, heap is not suitable if items must be unique. Skip-list, on the other hand, can be used even if items are not unique, like our implementation, or if duplicates are allowed, by slight modification (e.g., by adding internal identifiers to each node).

Herlihy and Koskinen's pessimistically boosted priority queue uses a concurrent heap-based priority queue. A global readers/writer lock is used on top of this priority queue to maintain semantic concurrency. An `add` operation acquires a read lock, while `getMin` and `removeMin` operations acquire a write lock. Thus, all `add` operations will be concurrently executed at the semantic level because they are semantically commutative. Global locking is a must here, because the `removeMin` operation is not commutative with either another `removeMin` operation or an `add` operation with a smaller item. Recall that, commutativity between operations is necessary for boosting.

Algorithm 9 shows the flow of pessimistic boosted operations (more details are in [57]). It is important to notice that the inverse of the `add` operation is not defined in most priority queue implementations. This is one of the drawbacks of pessimistic boosting, which cannot be implemented without defining an inverse for each operation. A work-around to this problem is to encapsulate each node in a holder class with a boolean `deleted` flag to mark rolled-back `add` operations (line 4). `removeMin` keeps polling the head until it reaches a non-deleted item (lines 8-10). This adds greater overhead to the boosted priority queue.

Algorithm 9 Pessimistic boosted priority queue.

```
1: procedure ADD($x$)
2:     readLock.acquire
3:     concurrentPQ.add($x$)
4:     undo-log.append(holder($x$).deleted = true)
5: end procedure

6: procedure REMOVEMIN
7:     readLock.acquire
8:     $x$ = concurrentPQ.removeMin()
9:     while holder($x$).deleted = true do
10:        $x$ = concurrentPQ.removeMin()
11:    undo-log.append(add($x$))
12: end procedure
```

Pessimistic boosting uses the underlying priority queue as a black box, either based on heap or skip-list or any other structure. This means that, the gains from using skip-list may be nullified by the pessimistic abstract locking. For example, since pessimistic boosting does not open the black box, if the underlying concurrent priority queue uses fine-grained locking to enhance performance, this optimization is hidden from the semantic layer and will be nullified by the coarse-grained semantic locking when non-committing operations execute concurrently. Optimistic boosting, on the other hand, inherits these benefits of using skip-list, and avoids eager locking. Since skip-list does not have a re-balance phase, we can implement an optimistic boosted priority queue based on it. However, before we show this version, we quickly describe how to extend TB to implement semi-optimistic heap-based priority queue in the following section.

4.2.1 Semi-Optimistic Heap Implementation

A semi-optimistic implementation of a heap-based priority queue is achieved by using the following three optimizations on top of the TB implementation (these optimizations are illustrated in Algorithm 10):

i) The `add` operations are not pessimistically executed until the first `removeMin` or `getMin` operation has occurred. Before that, all `add` operations are saved in a local semantic redo log. Once the first `removeMin` or `getMin` operation occurs, a single global lock on the whole data structure is acquired and all the local `add` operations stored in the redo log are published before executing the new `removeMin` operations. If only `add` operations occur in the whole transaction, they are published at commit time. This way, semi-optimistic boosting does not pay the overhead for managing read/write locks because `add` operations do not acquire any lock.

ii) Since the transaction that holds the global lock cannot be aborted by any other transaction, there is no need to keep the operations in a semantic undo log anymore. This is because, no operation takes effect on the shared objects until the global lock is acquired. This enhancement cannot be achieved in pessimistic boosting because `add` operations are executed eagerly on the shared object. Moreover, this optimization leads to further enhancement, because the guidelines of optimistic boosting relax the obligation of defining an inverse operation for the `add` operation. This way, the overhead of encapsulating each node in a holder class is avoided.

iii) In semi-optimistic priority queue, there is no need for thread-level synchronization, because no transaction accesses the shared object until it acquires the global semantic lock, which means that there is no contention on the underlying priority queue. Pessimistic boosting, on the other hand, has to use a concurrent priority queue, because `add` operations are executed eagerly and can conflict with each other.

Algorithm 10 Optimistic boosted heap-based priority queue.

```
procedure ADD($x$)
    if lock holder then
        PQ.add($x$)
    else
        redo-log.append($x$)
end procedure

procedure REMOVEMIN
    Lock.acquire
    for entries in redo-log do
        PQ.add(entry.item)
    $x$ = PQ.removeMin()
end procedure
```

The same idea of our enhancements has been used before in the TML algorithm [24] for memory-based transactions. In TML, a transaction keeps reading without any locking and defers acquiring the global lock until the first write occurs (which maps to the first `removeMin` operation in our case). It then blocks any other transaction from committing until it finishes its execution.

Although these optimizations diminish the effect of global locking, it cannot be considered as an optimistic implementation because `removeMin` and `getMin` operations acquire the global write lock before commit time (this is why we call it a "semi-optimistic" approach). In the next section, we will show how we can use skip-list to implement a completely optimistic priority queue.

4.2.2 Skip-List OTB Implementation

In this version, optimistic boosted priority queue wraps the same optimistic boosted skip-list described in Section 4.1. The same idea of using skip-list is proposed for a skip-list-based concurrent priority queue. However, implementation details are different here because of the new transactional nature of the boosted priority queue (more details about the concurrent implementation can be found in [59]).

Slight modifications are made on the optimistic boosted skip-list to ensure priority queue properties. Specifically, each thread saves, locally, a variable called $lastRemovedMin$, which refers to the last element removed by the transaction. It is mainly used to identify the next element to be removed if the transaction calls another `removeMin` operation (note that all these operations do not physically change the underlying skip-list until commit is called). Thus, this variable is initialized as the head of the skip-list. Additionally, each thread saves a local sequential priority queue, in addition to the local read/write sets, to handle read-after-write cases.

Algorithm 11 shows the wrapped priority queue. Each `add` operation calls the `add` operation of the underlying (optimistic boosted) skip-list. If it is a successful `add`, it saves the added value in the local sequential priority queue (line 3). Any `removeMin` operation compares the minimum items in both the local and shared priority queues and removes the lowest (line 11). If the minimum is the local one, the transaction calls the underlying `contains` operation to make sure that the shared minimum is saved in the local read-set to be validated at commit (line 12). Before returning the minimum, the transaction does a post-validation to ensure that the shared minimum is still linked by its *pred* (lines 14 and 20). It then updates $lastRemovedMin$ (line 22). A similar procedure is used for the `getMin` operation.

Using this approach, optimistic boosted priority queue (based on skip-list) does not acquire any locks until its commit. It follows the same general concepts of optimistic boosting, including lazy commit, post validation, and semantic read-sets and write-sets. Unfortunately, the same approach cannot be used in a heap-based implementation because of its complex and cooperative re-balancing mechanism. However, we already discussed a semi-optimistic heap-based implementation in the previous section.

One of the main advantages of this optimistic implementation is that the `getMin` operation is again wait-free. Pessimistic boosting, even with our enhancements on the heap-based implementation, enforces `getMin` to acquire a write lock, thereby becoming a blocking operation,

Algorithm 11 Optimistic boosted skip-list-based priority queue.

```
procedure ADD($x$)
    if skipList.add($x$) then
        localPQ.add($x$)
        return true
    else
        return false
end procedure

procedure REMOVEMIN
    $localMin$ = localPQ.getMin()
    $sharedMin$ = $lastRemovedMin$.next[0]
    if $localMin$ ¡ $sharedMin$ then
        if ¬ skipList.contains($sharedMin$) then
            Abort
        if $lastRemovedMin$.next[0] $\neq$ $sharedMin$ then
            Abort
        return localPQ.removeMin()
    else
        if ¬ skipList.remove($sharedMin$) then
            Abort
        if $lastRemovedMin$.next[0] $\neq$ $sharedMin$ then
            Abort
        $lastRemovedMin$ = $sharedMin$
        return $sharedMin$
end procedure
```

even for non-conflicting `add` or `getMin` operations.

4.2.3 Correctness

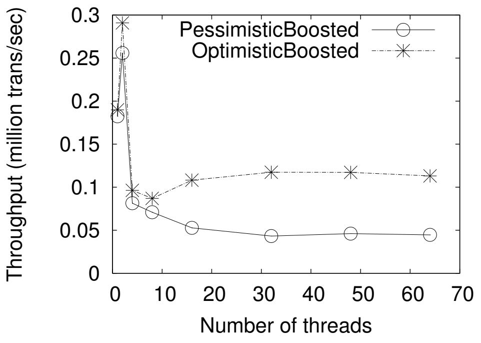
Correctness of the priority queue is easier to show than for the set. For the heap-based implementation, `add` operations are commutative because duplicates are allowed. This is the reason why deferring `add` operations to the lock acquisition phase does not affect consistency. Transactions will not be able to execute `removeMin` operations until they acquire the global lock and publish earlier `add` operations. The global locking yields the same correctness and progress guarantees of pessimistic boosting.

For the skip-list implementation, lock acquisition and validation are inherited from the underlying skip-list (the calling of `add`, `remove`, and `contains` adjusts the local read-set and write-set of each transaction). Additionally, the local priority queue is used to save the previously added items to maintain the case of getting or removing a locally added minimum. This follows the same approach of searching the local write-set in STM algorithms to cover the cases of read-after-write.

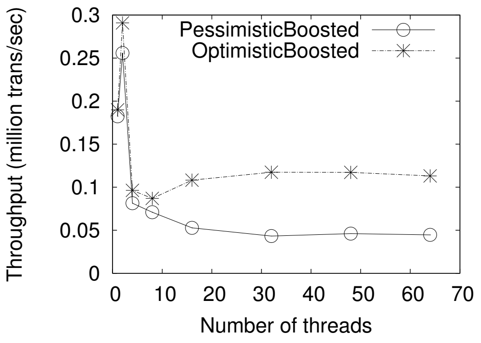
4.2.4 Evaluation

We now show the performance of heap-based priority queue and illustrate how our semi-optimistic implementation enhances performance. Then, we discuss the skip-list-based priority queue, and compare the performance of pessimistic and optimistic implementations. The testing environment is the same as in Section 4.1.4

For heap-based implementations, we used Java atomic package's priority queue for pessimistic boosting implementation, and Java's sequential priority queue for semi-optimistic boosting implementation. Figure 4.4 illustrates how our three optimizations (described in Section 4.2) enhance performance with respect to the pessimistic boosting algorithm. Since both `min` and `removeMin` operations acquire global lock, they will have the same effect on performance. This is why, for brevity, we only show results for workloads with 50% `add` operations and 50% `removeMin` operations. However, we also conducted experiments with different percentage of `getMin` operations and obtained similar results.



(a) Transaction size = 1

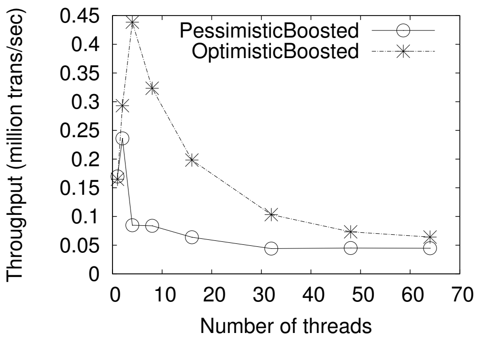


(b) Transaction size = 5

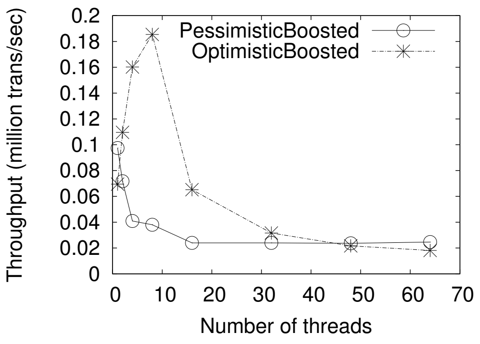
Figure 4.4: Throughput of heap-based priority queue with 512 elements, for two different transaction sizes (1, 5). Operations are 50% add and 50% removeMin.

The results show that our semi-optimistic boosting implementation is faster than pessimistic boosting irrespective of the transaction size (1 or 5 operations). Increasing the transaction size to 5 affects the performance of both algorithms, because both acquire global locks when the first `removeMin` occurs. However, semi-optimistic boosting is 2x faster than pessimistic boosting when transaction size is 5.

For skip-list-based priority queue, we use our skip-list set implementation (Section 4.1) as the base of priority queue implementation. Figure 4.5 shows the performance of both optimistic and pessimistic boosting. Optimistic boosting is better than pessimistic boosting in almost all the cases, except for the high contention case (5 operations per transaction and more than 48 transactions). Comparing the results in Figures 4.4 and 4.5, we see that optimistic boosting achieves the best performance with respect to all other algorithms (both heap-based and skip-list-based) for small number of threads. This improvement is achieved at the cost



(a) Transaction size = 1



(b) Transaction size = 5

Figure 4.5: Throughput of skip-list-based priority queue with 512 elements, for two different transaction sizes (1, 5). Operations are 50% add and 50% removeMin.

of a slightly lower performance when the number of transactions increases. This is expected, and reasonable for optimistic approaches in general, given that the gap in performance for high contention cases is limited.

4.3 Summary

In this chapter we provided a detailed design and implementation of two representative optimistically boosted data structures: set and priority queue, and we showed how the same concept of optimistic boosting can be applied to different implementations of these data structures. Our evaluation revealed that the performance of optimistic boosting is closer to highly concurrent data structures than pessimistic boosting in most of the cases.

Chapter 5

Transactional Balanced Trees

Balanced binary search trees, such as AVL and Red-Black trees are data structures whose self-balancing guarantees an appealing logarithmic-time complexity for their operations. One of the main issues in balanced trees is the need for *rotations*, which are complex housekeeping operations that re-balance the data structure to ensure its logarithmic-time complexity. Although rotations complicate the design of concurrent balanced trees, many solutions have already been proposed: some of them are lock-based [13, 21, 22, 5, 8, 34], while others are non-blocking [15, 35, 63, 84]. However, none of those approaches allows tree operations to compose.

We leverage OTB methodology and design TxCF-Tree, the first balanced tree that is accessible in a *transactional*, rather than just a *concurrent*, manner without monitoring (speculating) the whole traversal path (like in TM) or nullifying the benefits of the efficient concurrent designs (like in TB). TxCF-Tree offers a set of design and low-level innovations, but roughly it can be seen as the transactional version of the recently introduced *Contention Friendly Tree* (CF-Tree) [22]. The main idea of CF-Tree is to decouple the *structural operations* (e.g., rotations and physical deletions) from the *semantic operations* (e.g., queries, logical removals, and insertions), and to execute those structural operations in a dedicated *helper* thread. This separation makes the semantic operations (that need to be transactional in TxCF-Tree) simple: each operation traverses the tree non-speculatively (i.e., without instrumenting any accessed memory location); then, if it is a write operation, it locks and modifies only one node. In an abstract way, the TxCF-Tree's semantic operations can be seen as composed of a *traversal* and *commit* phases, which makes CF-Tree a good representative of the OTB-based data structures.

In addition to the new transactional capabilities, TxCF-Tree claims one major innovation with respect to CF-Tree, which is fundamental for targeting high performance in a transactional (not only concurrent) data structure. Although CF-Tree decouples the structural operations, those operations are executed in the *helper* thread with the same priority as the semantic operations, and without any control on their interference. With TxCF-Tree, we make the structural operations *interference-less* (when possible) with respect to semantic operations. This property is highly desirable because structural operations do not alter the abstract (or semantic) state of the tree, thus they should not force any transaction to abort. To reduce this interference, one operation should behave differently if it conflicts with a structural operation rather than with a semantic operation.

TxCF-Tree uses two new terms, which help to identify those *false-interleaving* cases and alleviate their effect: *structural lock*, which is a type of lock acquired if the needed modifications on the node do not change its abstract (semantic) state; and *structural invalidation*, which is a transactional invalidation raised only because of a structural modification on the tree rather than having actual conflicts at the abstract level. In TxCF-Tree, transactions do not abort if they face structural locks or false-invalidations during the execution of their operations. We further reduce the interference of the *helper* thread by adopting a simple heuristic to detect if the tree is *almost balanced*. If so, we increase the back-off time between two *helper* thread's iterations.

We assessed the effectiveness of TxCF-Tree through an evaluation study. Our experiments show that TxCF-Tree performs better than the other transactional approaches (TB and STM) in almost all of the cases.

5.1 Background: Contention Friendly Tree

Contention Friendly Tree (CF-Tree) [22] is an efficient concurrent lock-based (internal) tree, which finds its main innovation on decoupling the semantic operations (i.e., search, logical deletion, and insertion) from the structural operations (i.e., rotation and physical deletion). The semantic operations are eagerly executed in the original process, whereas the structural operations are deferred to a helper thread. More in details:

Semantic Operations: each semantic operation starts by traversing the tree until it reaches a node that matches the requested key or it reaches a leaf node (indicating that the searched node does not exist). After that, a search operation returns immediately with the appropriate result without any locking. For a deletion, if the node exists and it is not marked as deleted, the node is locked and then the *deleted* flag is set (only a logical deletion), otherwise the operation returns false. For a successful insertion, the *deleted* flag is cleared (if the node already exists but marked as *deleted*) or a new node is created and linked to the leaf node (if the node does not exist). An unsuccessful insertion simply returns false. In all cases, each operation locks at most one node.

Rotations: re-balancing operations are isolated in a *helper* thread that scans the tree seeking for any node that needs either a rotation or a physical removal. Rotation in this case is relaxed, namely it uses local heights. Although other threads may concurrently modify these heights (resulting in a temporarily unbalanced tree), past work has shown that a sequence of localized operations on the tree eventually results in a strictly balanced tree [13, 69]. A rotation locks: the node to be rotated down; its parent node; and its left or right child (depending on the type of rotation). Also, rotations are designed so that any concurrent semantic operation can traverse the tree without any locking and/or instrumentation. To achieve that, the rotated-down node is cloned and the cloned node is linked to the tree instead of the original node.

Physical Deletion: The physical deletion is also decoupled and executed separately in the *helper* thread. In addition, a node's deletion is relaxed by leaving a "routing" node in the tree when the deleted node has two children (it is known that deleting a node with two children requires modifying nodes that are far away from each other, which complicates the operation). The physical deletion is done as follows: both the deleted node and its parent are locked, then the node's left and right children links are modified to be pointing at its parent, and finally the node is marked as *physically removed*. This way, concurrent semantic operations can traverse the tree non-speculatively without being lost.

Among the concurrent trees presented in literature, we select CF-Tree as a candidate to be transactionally boosted because it provides the following two properties that fit the OTB principles. First, it uses a lock-based technique for synchronizing the operations, which simplifies the applicability of the OTB methodology. Second, CF-Tree is traversed without any locking and/or speculation, allowing the separation of an unmonitored traversal phase. Also, the semantic operations (`add`, `remove`, and `contains`) are decoupled from the complex structural operations (although they can interfere with each other), like rotations and physical removals, allowing a simple commit phase.

5.2 Reducing the interference of structural operations

Balanced trees store data according to a specific balanced topology so that their operations can take advantage of the efficient logarithmic-time complexity. More specifically, operations are split into two parts: a "semantic" part, which modifies the abstract state of the tree, and a structural part, which maintains the efficient organization of the tree. For example, consider the balanced tree in Figure 5.1[1]. The tree initially represents the abstract set $\{1, 2\}$ (Figure 5.1(a)). If we want to insert 3, we first create a new node and link it to the tree in the proper place (Figure 5.1(b)). Subsequently, the tree is re-balanced because this insertion unbalanced a part of it (Figure 5.1(c)). Semantically, we can observe the new abstract set, $\{1, 2, 3\}$, right after the first step and before the re-balancing step. However, without the re-balancing step, the tree structure itself may become eventually skewed, and any traversal operation on the tree would take linear time rather than logarithmic time.

Although the structural operations are important, like the aforementioned rotations in our case, they represent the main source of conflicts when concurrent accesses on the tree occur. Two independent operations (like inserting two nodes in two different parts of the

[1]We assume that higher keys are in the left sub-tree to match CF-Tree's design.

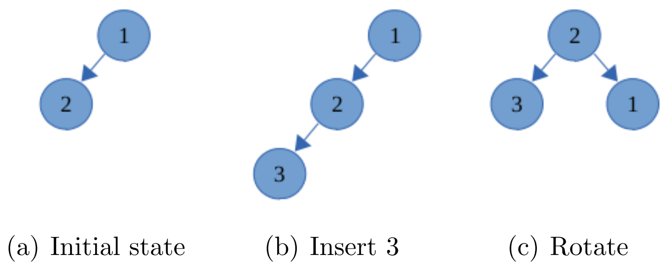


Figure 5.1: An insertion followed by a right rotation in a balanced tree.

tree) may conflict only because one of them needs to re-balance the tree. This additional conflict generated by structural operations can significantly slow down the performance of transactional data structures more than their concurrent versions due to two reasons. First, in long transactions, the time period between the tree traversal and the actual modification during commit may be long enough to generate more conflicts because of the concurrent re-balancing. Second, in transactional data structures, any conflict can result in the abort and re-execution of the whole transaction, which possibly includes several non-conflicting operations, unlike concurrent operations that just re-traverse the tree if a conflict occurs.

Although CF-Tree decouples the structural operations in a dedicated *helper* thread, which forms an important step towards shortening the critical path of the processing (i.e., the semantic operations), it does not prevent the structural operations running in the *helper* thread from interfering with the semantic operations and delaying/aborting them. To minimize such a interference, we propose the following simple guideline (named *G-Pr*), which can be added to the general guidelines of OTB presented in Section 3.2:

"*Semantic operations should have higher priority than structural operations.*"

This guideline allows semantic operations to proceed if a conflict with structural modifications occurs. Our rationale is that, delaying (or aborting) semantic operations affects the performance, whereas delaying (or aborting) structural operations only defers the step of optimizing the tree to the near future.

5.3 TxCF-Tree

In this section, we discuss how to boost CF-Tree to be transactional using the OTB guidelines. The key additions of TxCF-Tree over CF-Tree are: *i)* supporting transactional accesses; and *ii)* minimizing the interference between semantic and structural operations. to simplify the presentation, we focus on the changes made on CF-Tree to achieve those two goals, and we briefly mention the unchanged parts whose details can be found in [22].

Each node in TxCF-Tree contains the same fields as CF-Tree: a key (with no duplication allowed), two pointers to its left and right children, a boolean *deleted* flag to indicate the logical state of the node, and an integer *removed* flag to indicate the physical state of the node (a value from the following: `NOT-REMOVED`, `REMOVED`, or `REMOVED-BY-LEFT-ROTATION`). The node structure in TxCF-Tree is only different in the locking fields. In CF-Tree, each node contains only one lock that is acquired by any operation modifying the node. In TxCF-Tree, each node has two different locks: a *semantic-lock*, which is acquired by the operations that modify its semantic state (either the *deleted* or the *removed* flag); and a *structural-lock*, which is a acquired by the operations that modify the structure of the tree without affecting the node itself (i.e., modifying the right or left pointers). Each lock is associated with a *lock-holder* field that saves the ID of the thread that currently holds the lock, which is important to avoid deadlocks.

TxCF-Tree implements a *set* interface with the semantic operations: `add`, `remove`, and `contains`. Extending TxCF-Tree to have key-value pairs is simple, but for clarity we assume that the value of the node is the same as its key.

5.3.1 Structural Operations

The *helper* thread repeatedly calls a recursive depth-first procedure to traverse the entire tree. During this procedure, any unbalanced node is rotated and any logically removed node is physically unlinked from the tree. To minimize the interference of this *housekeeping* procedure, we use an adaptive back-off delay after each traversal iteration. We use a simple hill-climbing mechanism that increases (decreases) the back-off time if the number of housekeeping operations in the current iteration is less (greater) than the most recent iteration. While acknowledging the simplicity of the adopted heuristic, it showed effectiveness in our evaluation study.

Physical Deletions. We start by summarizing how the *helper* thread in CF-Tree physically deletes a node N_n (marked as *deleted* and at least one of its children is `null`). First, both N_n and its parent N_p are locked. Then, the node's left and right children fields are modified to point back to the parent (so that the concurrent operations currently visiting N_n can still traverse the tree, without experiencing any interruption) and then N_n is marked as `REMOVED` and unlinked by changing N_p child to be N_n's child instead of N_n.

TxCF-Tree modifies this mechanism by providing *less-interfering* locking. Specifically, we only acquire the *structural-lock* of N_p because its semantic state will not change. On the other hand, both the *semantic-lock* and the *structural-lock* have to be acquired on N_n because N_n's *removed* flag, which is part of its semantic state, should be set as `REMOVED`. To further minimize the interference, the locking mechanism uses only one CAS trial. If it fails, then the whole structural operation is aborted and the *helper* thread resumes scanning the tree.

Rotations. In CF-Tree, a right rotation (without losing generality) locks three nodes: the parent node N_p, the node to be rotated down N_n, and its left child N_l. Then, rotation is done by cloning N_n and linking the cloned node at the tree instead of N_n (similar to physical deletion, this cloning protects operations whose "unmonitored" traversal phase is concurrently visiting the same nodes. More details are in [22]). Subsequently N_n is marked as `REMOVED` (in case of left-rotation it is marked as `REMOVED-BY-LEFT-ROTATION`) and nodes are unlocked.

In TxCF-Tree, rotations also use a less intrusive locking mechanism. Both N_p and N_l acquire only the *structural-lock* because the rotated-down node N_n is the only node that will change its semantic state (and thus needs to acquire the *semantic-lock*). Also, we found that there is no need to lock the parent node (i.e., N_p) at all. This is because the only change to N_p is to make its left (or right) child pointing to N_l rather than N_n. This means that N_p's child remains not `null` before and after the rotation. Only the *helper* thread can change it to `null` in a later operation by rotating the node down or physically deleting its children. On the other hand, semantic operations only concern about reading/changing the *deleted* flag of a node, if the searched node exists in the tree, or reading/changing a (`null`) link of a node, if the searched node does not exist in the tree. Thus, modifying the child link of N_p cannot conflict with any concurrent semantic operation, thus it is safe to make this modification without locking. Similarly, if all the sub-trees of N_n and N_l are not `null`, then no structural locks are acquired, and the only lock acquired is the *semantic-lock* on N_n.

5.3.2 Semantic Operations

According to OTB, each operation is divided into the *traversal*, *validation*, and *commit* phases. We follow this division in our presentation.

Traversal. The tree is traversed by following the classical rules of the sequential binary search tree. Traversal ends if we reach the searched node or a `null` pointer. To be able to execute the operation transactionally, the outcome of the traversal phase is not immediately returned. Instead it is saved in a local semantic read/write sets. Each entry of those sets consists of the following three fields. *Op-key*: the searched key that needs to be inserted, removed, or looked up. *Node*: the last node of the traversal phase. This node is either a node whose key matches op-key (no matter if it is marked as *deleted* or not) or a node whose right (left) child is `null` and its item is greater (less) than op-key. *Op-type*: an integer that indicates the type of the operation (`add`, `remove`, or `contains`) and its result (`successful` or `unsuccessful`).

Those fields are sufficient to verify (by the transaction validation) that the result of the operation is not changed since the execution of the operation, and to modify the tree at commit time. All the operations add an entry to the read-set, but only successful `add` and `remove` operations add entries to the write-set.

Before traversal, the local write-set is scanned for detecting read-after-write hazards. If the key exists in the write-set, the operation returns immediately without traversing the shared tree. Moreover, if a successful `add` operation is followed by a successful `remove` operation of the same item (or vice versa), they locally eliminate each other, in order to save the useless access to the shared tree. The elimination is done only on the write-set, and the entries are kept in the read-set so that the eliminated operations are guaranteed to be consistent.

Validation. The second phase of TxCF-Tree's operation is the *validation* phase. To have a comprehensive presentation, we show first the validation procedure in CF-Tree, and then we show how it is modified in TxCF-Tree.

Algorithm 12 Operation's validation in CF-Tree.

```
 1: procedure VALIDATE(node, k)
 2:     if node.removed ≠ NOT-REMOVED then
 3:         return false
 4:     else if node.k = k then
 5:         return true
 6:     else if node.k > k then
 7:         next = node.right
 8:     else
 9:         next = node.left
10:     if next = null then
11:         return true
12:     return false
13: end procedure
```

In Algorithm 12, the validation in CF-Tree succeeds if the node's key is not physically removed and either the node's key matches the searched key (line 5) or its child (right or left according to the key) is still `null` (line 11). Otherwise, the validation fails (lines 3 and 12). This validation is used during `add/remove` operations as follows (details are in [22]): each operation traverses the tree until it reaches the involved node, then it locks and validates it (using Algorithm 12). If the validation succeeds, the operation stops its traversal loop and starts the actual insertion/deletion. If the validation fails, the node is unlocked and the operation continues the traversal. In [22], it has been proven that continuing the traversal is safe even if the node is physically deleted or rotated by the *helper* thread, due to the mechanism used in the deletion/rotation, as discussed in Section 5.3.1 (e.g., modifying the left and right links of the deleted node to be pointing to its parent before unlinking it).

Algorithm 13 Example of semantic opacity.

```
 1: @Atomic                                  ▷ initially the tree is empty
 2: procedure T1
 3:     if tree.contains(x) = false then
 4:         if tree.contains(y) = true then
 5:             ...                          ▷ hazardous action
 6: end procedure
 7: @Atomic
 8: procedure T2
 9:     tree.add(x)
10:     tree.add(y)
11: end procedure
```

In TxCF-Tree, this validation procedure is modified to achieve two goals.
The first goal regards the correctness: since TxCF-Tree is a transactional tree, validation has also to ensure that the operation's result is not changed until transaction commits; otherwise, the transaction consistency is compromised. As an example, in Algorithm 13 let us assume the following invariant: y exists in the tree if and only if x also exists. If we use the same validation as Algorithm 12, $T1$ may execute line 3 first and return false. Then, let us assume that $T2$ is entirely executed and committed. In this case, $T1$ should abort

right after executing line 4 because it breaks the invariant. Aborting the doomed transaction $T1$ should be immediate and it cannot be delayed until the commit phase because it may go into an infinite loop or raise an exception (line 5). To prevent those cases, all of the read-set's entries have to be validated (using Algorithm 14 instead of Algorithm 12) after each operation as well as during commit.

Algorithm 14 Operation's validation in TxCF-Tree.

```
 1: procedure VALIDATE(read-set-entry)
 2:     if entry.op-type $\in$ (unsuccessful add,
 3: successful remove/contains) then
 4:         item-existed = true
 5:     else
 6:         item-existed = false
 7:     if entry.node.removed $\neq$ NOT-REMOVED then
 8:         return STRUCTURALLY-INVALID
 9:     else if entry.node.k = entry.op-key then
10:         if entry.node.deleted xor
11: item-existed then
12:             return VALID
13:         else
14:             return SEMANTICALLY-INVALID
15:     else if entry.node.k $>$ entry.op-key then
16:         next = node.right
17:     else
18:         next = node.left
19:     if next = null then
20:         if item-existed then
21:             return SEMANTICALLY-INVALID
22:         else
23:             return VALID
24:     return STRUCTURALLY-INVALID
25: end procedure
```

The second goal regards performance: if the node is physically removed or its child becomes no longer `null` (which are the invalidation cases of CF-Tree), that does not mean that the transaction is not consistent anymore. It only means that the traversal phase has to continue and reach a new node to be validated. It is worth noting that aborting the transaction in those cases does not impact the tree's correctness, while its performance will be affected. In fact, this conservative approach increases the probability of structural operations' interference. For this reason we distinguish between those types of invalidations and the actual *semantic* invalidations, such as those depicted in Algorithm 13. The modified version of the validation is shown in Algorithm 14. The cases covered in CF-Tree are considered *structural-invalidations* (lines 8 and 24), and the actual invalidation cases are considered *semantic-invalidations* (lines 14 and 21).

Algorithm 15 Read-set validation in TxCF-Tree.

```
 1: procedure VALIDATE-READSET(read-set)
 2:     for all entries in the read-set do
 3:         while true do
 4:             if entry.op-key = entry.node.k then
 5:                 lock = semantic-lock
 6:             else
 7:                 lock = struct-lock
 8:             if lockedNotByMeOrHelper(lock) then
 9:                 return false
10:             r = VALIDATE(entry)
11:             if r = STRUCTURALLY-INVALID then
12:                 newNode = CONT-TRAVERSE(entry)
13:                 entry.node = newNode
14:                 write-entry = write-set.get(entry.op-key)
15:                 if write-entry $\neq$ null then
16:                     write-entry.node = newNode
17:             else if r = SEMANTICALLY-INVALID then
18:                 return false
19:             else
20:                 break;
21:     return true
22: end procedure
```

Algorithm 15 shows how to validate the read-set. For each entry, we firstly check if the entry's node is not locked (lines 4-9). In this step we exploit our lock separation by checking only one of the two locks because each operation validates either the *deleted* flag or the child link. Specifically, if the node's key matches op-key, node's *semantic-lock* is checked, otherwise the *structural-lock* is checked. Moreover, if the entry's node is locked by the *helper* thread, we consider it as unlocked because the helper thread cannot change the abstract state of the tree. The only effect of the *helper* thread is to make the operation structurally invalid, which can be detected in the next steps.

The next step is to validate the entry itself (line 10). If it is *semantically-invalidated*, then the transaction aborts (line 18). If it is *structurally-invalidated*, the traversal continues as in CF-Tree and the entry is updated with the new node (lines 12-16), then the node is re-validated. If the operation is a successful *add/remove*, the related write-set entry is also updated (line 16).

Commit. The *commit* phase (Algorithm 16) is similar to the classical two-phase locking mechanism. The nodes in the read/write sets are locked and/or validated first, then the tree is modified, and finally locks are released.

From the commit procedure of TxCF-Tree it is worth mention the following points. The first point is how TxCF-Tree solves the issue of having two dependent operations in the same transaction. For example, if two *add* operations are using the same node (e.g., assume a transaction that adds both 3 and 4 to the tree shown in Figure 5.1). The effect of the first operation (add 3) should be propagated to the second one (add 4). To achieve that, the add operation uses the node in the write-set only as a starting point and keeps traversing the tree from this node until reaching the new node. Also, the operations lock the added nodes (3 and 4 in our case) before linking them to the tree. Those nodes are unlocked together with the other nodes at the end of the commit phase. Any interleaving transaction or structural operation running in the *helper* thread cannot force the transaction to abort because all the involved nodes are already locked. Also, the other cases of having dependent operations, such as adding (or removing) the same key twice and adding a key and then removing it, are solved earlier during the operation itself (as mentioned in the traversal phase).

The second point is how TxCF-Tree preserves the reduced interferences between the structural and the semantic operations without hampering the two-phase locking mechanism. The main issue in this regard is that *structural invalidations* may not abort the transaction. Thus, a transaction cannot lock the nodes in the write-set and then validate the nodes in the read-set because, if so, in case of a *structural invalidation*, the invalidated operation (which can be a write operation) would continue traversing the tree and reach a new node (which is not yet locked). To solve this problem, we use an *inline* validation of the entries in the write-set. The write-set entries are both locked and validated at the same time. If the write operation fails in its validation: *1)* it unlocks the node; *2)* re-traverses the tree; *3)* locks the new node; and *4)* re-validates the entry.

Algorithm 16 Commit in TxCF-Tree.

```
procedure COMMIT
    for all entries in the write-set do
        while true do
                                                        ▷ Try to acquire the lock
            if entry.op-key = entry.node.k then
                lock = semantic-lock
            else
                lock = struct-lock
            if lockholder ≠ myID and !lock.acquire then
                if lockholder ≠ helperID then
                    ABORT
                else
                    continue
                                                        ▷ Inline Validation
                                                        ▷ Similar to Algorithm 15
                                                        ▷ But unlock before retrying
            result = VALIDATE(entry)
            ...
                                                        ▷ Validate the remaining read-set entries
                                                        ▷ Exactly like Algorithm 15
                                                        ▷ But skips the entries that are also in the write-set
        VALIDATE-READ-OPERATIONS(read-set)
                                                        ▷ Publish write-sets
        for all entries in the write-set do
            if entry.op-type = remove then
                entry.node.deleted = true
            else                                        ▷ add operation
                if entry.op-key = entry.node.k then
                    entry.node.deleted = false
                else
                    newNode = CREATE-NODE(entry.key)
                    node = CONT-TRAVERSE(entry)
                    if node.key > entry.k then
                        node.right = newNode;
                    else
                        node.left = newNode;
                                                        ▷ Unlock
        UNLOCK(write-set)
    return true
end procedure
```

5.4 Correctness

Similar to OTB-Set, the correctness of TxCf-Tree can be inferred by the following two steps. The first step is to show, without loss of generality, that if the transactions are composed of only one TxCF-Tree operation, then these operations are linearizable. The second step is to prove that if the transaction contains more than one operation, then it is opaque. We can easily achieve that by following the same procedure described in Section 4.1.3. Thus, in this section we focus on the points specifically related to TxCF-Tree.

Since we use the OTB methodology to make CF-Tree transactional, the operations of TxCF-Tree are serialized as described in [61]rialization point of a read-write transaction is the point right after acquiring the locks and before the (successful) validation during commit. For a read-only transaction, the serialization point is the return of its last read operation. Both those points are immediately followed by a validation procedure (Algorithm 15). If this validation succeeds, then all the transaction operations are guaranteed to be consistent.

The correctness of the mechanisms used to achieve *interference-less* structural operations can be inferred as follows.

i) Splitting locks into *structural* and *semantic* locks does not affect correctness by any mean, because any two conflicting operations (e.g., two operations that attempt to delete the same node, or two operations that attempt to insert new nodes on the same link) acquire the same type of lock.

ii) *Structural invalidations* are raised and handled in the same way as CF-Tree (as we show in Algorithms 12 and 14). Since we use the same approach for rotation and physical deletion (e.g., cloning the rotated down node and linking the physically deleted node to its parent), re-traversing the tree after a *structural invalidation* is guaranteed to be safe as in CF-Tree itself (see [22] for the complete proof of validation in CF-Tree).

iii) *Semantic invalidations* preserve the consistency among the operations within the same transaction. Unlike *structural invalidations*, in those cases, the whole transaction is aborted.

iv) The *inline* validation during commit does not affect the correctness (although it violates two-phase locking) because every *inline-validated* node is locked before being validated and cannot be invalidated anymore if the validation succeeds.

v) Validating the whole read-set after each operation and before committing preserves consistency in the presence of concurrent structural operations. For example, assuming the scenario where a structural operation physically removes a node that is used by a running transaction T1, which can be followed by a semantic operation (executed in another transaction T2) that adds this node in a different place of the tree. Although this new addition will not be detected by T1's validation, the expected race condition will be solved because T1 will detect during the validation (after the next operation or at commit) that the *removed* flag of the node has been changed (line 7 in Algorithm 14) and will continue traversing the tree. At this point, T1 will reach the same new node as T2, and they will be serialized independently from the structural operation.

It is clear that those serialization points are not sufficient to guarantee opacity at memory level (i.e., in a history composed of all the memory locations accessed while performing the semantic operations). This is mainly because each operation in TxCF-Tree traverses the tree non-speculatively and all the reads during this traversal phase can be invalidated by any concurrent transaction. However, our target is to make TxCF-Tree *semantically* consistent.

5.5 Evaluation

In our experiments we compared the performance of TxCF-Tree with the performance of TB and some STM approaches. Our implementation of TB uses CF-Tree as the underlying (black-box) tree, which makes a fair comparison. Regarding STM, we tested three different algorithms: LSA [89]; TL2 [31]; and NOrec [25], and, to make plots clear, we reported the best performance collected.

All experiments were conducted on a 64-core machine, which has 4 AMD Opteron (TM) Processors, each with 16 cores running at 1.4 GHz, 32 GB of RAM, and 16KB L1 data cache. Throughput is measured as the number of semantic operations (not transactions) per second to have consistent data points. However, since the benchmark executes 256 `no-op` instructions in between two transactions, this may result in different throughput ranges for different sizes of transactions. Each data point is the average of five runs.

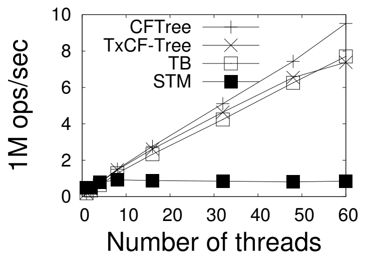
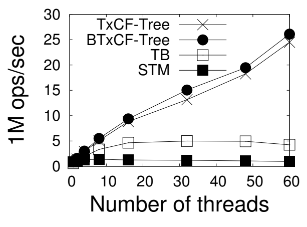


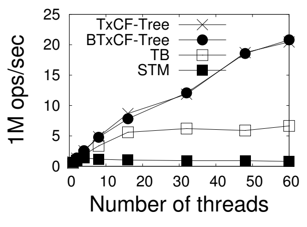
Figure 5.2: Throughput of tree-based set with 10K elements, 50% `add/remove` operations, and one operation per transaction.

In Figure 5.2 we show the results for a scenario that mimics the concurrent (non-transactional) case (i.e., each transaction executes only one operation on the tree). We leverage this plot to show the cost of adopting a transactional solution over a pure concurrent tree. Clearly STM does not scale because it "blindly" speculates on all the memory reads and writes. This poor scalability of STM is confirmed in all the experiments we made. On the other hand, both TB and TxCF-Tree scale better than STM and close to CF-Tree (TxCF-Tree is slightly closer). This behavior shows an overhead that is affordable in case one wants to use the TxCF-Tree library even for just handling the concurrency of atomic semantic operations without transactions.

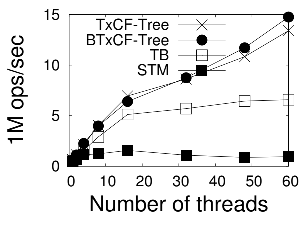
Figure 5.3 shows the transactional case, in which we deployed five operations per transaction for different sizes of the tree (1K, 10K, and 100K) and different read/write workloads (10%, 50% and 80% of `add/remove` operations). We do not include CF-Tree because it only supports concurrent operations and thus it cannot handle the execution of transactions. TxCF-Tree performs generally better than TB. The gap between the two algorithms decreases when we increase the percentage of the write operations. This is reasonable because the conflict level becomes higher, and it best fits the more *pessimistic* approach (as TB).



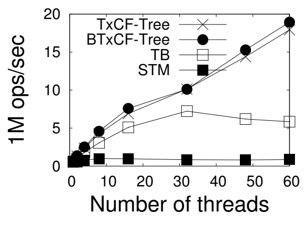
(a) 1K, 10%



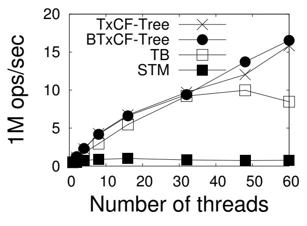
(b) 10K, 10%



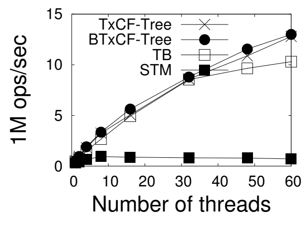
(c) 100K, 10%



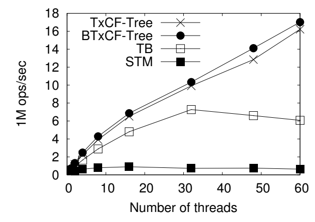
(d) 1K, 50%



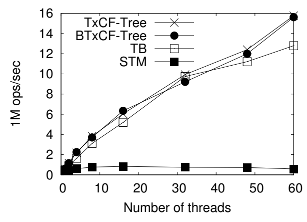
(e) 10K, 50%



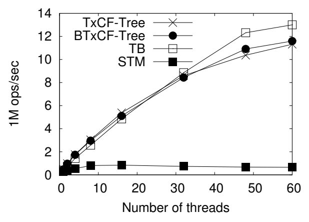
(f) 100K, 50%



(g) 1K, 80%



(h) 10K, 80%



(i) 100K, 80%

Figure 5.3: Throughput with five operations per transaction (labels indicate the size of the tree and the % of the `add/remove` operations).

Increasing the size of the tree also decreases the gap between TxCF-Tree and TB. At first impression it appears counterintuitive because increasing the size of the tree means generally decreasing the overall contention, which should be better for optimistic approaches like TxCF-Tree. The actual reason is that, in the case of very low contention, most of the transactions do not conflict with each other and both algorithms linearly scale. Then, when the conflict probability increases, the difference between the algorithms becomes visible. A comparison between Figure 5.3(d) and Figure 5.3(f) (which differ only for the size of the tree) confirms this claim. In Figure 5.3(d), both algorithms scale well up to 32 threads because threads are almost non-conflicting, then TB starts to suffers from its non-optimized design while TxCF-Tree keeps scaling. On the other hand, in Figure 5.3(f) both algorithms scale until 60 threads because the tree is large.

Summarizing, analyzing the above results we can identify two points that allow TxCF-Tree to outperform competitors: *i)* having an optimized unmonitored traversal phase that reduces false conflicts, and *ii)* having optimized validation/commit procedures that minimize the interferences between structural and semantic operations. Both TB and TxCF-Tree

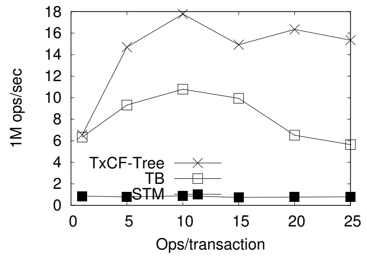


Figure 5.4: Throughput of tree-based set with 10K elements, 50% `add/remove` operations, and 32 threads.

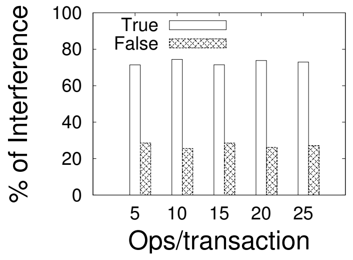


Figure 5.5: Percentage of the two interference types with 10K elements, 50% `add/remove`, and 32 threads.

gain performance by exploiting the first point, in fact TB itself performs (up to an order of magnitude) better than STM. However, only TxCF-Tree uses an *optimized* design for a balanced tree data structure, and it makes its performance generally (much) better than TB. In the aforementioned experiments we use two versions of TxCF-Tree, one with the adaptive back-off time in between two *helper* thread iterations (named BTxCF-Tree), and one without. The results show that this optimization further enhances the performance, especially in the small tree (the cases of 10% `add/remove` operations). This gain may increase with a more effective heuristic.

In Figure 5.4 we report the behavior of TxCF-Tree's while changing the size of the transactions. We can observe a significant gap between TxCF-Tree and TB for all of the tested sizes, which confirms our conclusion: reducing operations' interference is important in order to avoid unnecessary aborts.

The last experiment we report regards the capability of TxCF-Tree to reduce interferences with structural operations. Although breaking down TxCF-Tree's operations to measure this gain is not straightforward, we roughly estimated the gain by quantifying two metrics: the *true* interferences count, which is simply the actual transactional aborts count; and the *false* interferences count, which is the count of the cases in which the transaction does not abort because the tree is re-traversed instead or because the operations in TxCF-Tree acquire only one (structural or semantic) lock. In Figure 5.5 the false-interferences are 25%-30% of the total interferences for different sizes of the transactions.

5.6 Summary

In this chapter we presented TxCF-Tree, the first interference-less transactional balanced tree. Unlike the former general approaches, it uses an optimized conflict management mechanism that reacts differently according to the type of the operation. Our experiments confirm that TxCF-Tree performs better than the general approaches.

Chapter 6

Integrating OTB with STM

All previous proposals to implement transactional data structures, including TB, do not give details on how to integrate the proposed transactional data structures with STM frameworks. Addressing this issue is important because it allows programmers to combine operations of the efficient transactional data structures with traditional memory reads/writes in the same transaction.

In this chapter, we show how to integrate OTB-Based data structures with the current STM frameworks. One of the main benefits of OTB (compared to the original TB methodology) is that it uses the terms validation and commit in the same way as many STM algorithms [25, 31], but at the semantic layer. Thus, OTB allows building a system which combines both semantic-based and memory-based validation/commit techniques in a unified consistent framework. More specifically, we show in this chapter how to implement OTB data structures in a standard way that can integrate with STM frameworks. We also show how to modify STM frameworks to allow such integration while maintaining the consistency and programmability of the framework.

We use DEUCE [67] as our base framework. DEUCE is a Java STM framework with a simple programming interface. It allows users to define *@Atomic* functions for the parts of code that are required to be executed transactionally. However, like all other frameworks, using transactional data structures inside *@Atomic* blocks requires implementing pure STM versions, which dramatically degrades the performance. We extend the design of DEUCE to support OTB transactional data structures (with the ability to use the original pure STM way as well). To do so, we integrate two main components into the DEUCE agent. The first component is OTB-DS (or OTB data structure), which is an interface to implement any optimistically boosted data structure. The second component is OTB-STM Context, which extends the original STM context in DEUCE. This new context is used to implement new STM algorithms which are able to communicate with OTB data structures. The new STM algorithms should typically be an extension of the current memory-based STM algorithms in literature.

As a case study, we integrate our OTB-Set (as described in Chapter 4) in DEUCE framework. We extend two STM algorithms to communicate with OTB-Set (NOrec [25] and TL2 [31]). We select NOrec and TL2 as examples of STM algorithms which use different levels of lock granularity. NOrec is a coarse-grained locking algorithm, which uses a single global lock at commit time to synchronize transactions. TL2, on the other hand, is a fine-grained locking algorithm, which uses ownership records for each memory block. We show in detail how to make the extended design of DEUCE general enough to support both levels of lock granularity.

6.1 Extension of DEUCE Framework

DEUCE [67] is a Java STM framework which provides a simple programming interface without any additions to the JVM. It allows programmers to define *atomic* blocks, and guarantees executing these blocks atomically using an underlying set of common STM algorithms (e.g., NOrec [25], TL2 [31], and LSA [89]). We extend DEUCE to support calling OTB data structures' operations along with traditional memory reads and writes in the same transaction, without breaking transaction consistency. Our framework is designed in a way that integration between data structures' operations and memory accesses is completely hidden from the programmer. For example, a programmer can write an atomic block like that shown in Algorithm 2.

Figure 6.1 shows the DEUCE framework with the proposed modifications needed to support OTB integration. For the sake of a complete presentation, we briefly describe in Section 6.1.1 the original building blocks of the DEUCE framework (the white blocks with numbers 1-3). Then, in Section 6.1.2, we describe our additions to the framework to allow OTB integration (gray blocks with numbers 4-7). More details about the original DEUCE framework can be found in [67].

6.1.1 Original DEUCE Building Blocks

The original DEUCE framework consists of three layers:

Application layer. DEUCE applications do not use any new keywords or any addition to the language. Programmers need only to put an *@Atomic* annotation on the methods that they need to execute as transactions. If programmers want to include in the *@Atomic* blocks some operations that are not transactional by nature, like system calls and I/O operations, DEUCE allows that by using an *@Exclude* annotation. Classes marked as excluded are not instrumented by DEUCE runtime.

DEUCE runtime layer. Given this simple application interface (only *@Atomic* and *@Exclude* annotations), DEUCE runtime guarantees that atomic methods will be executed in

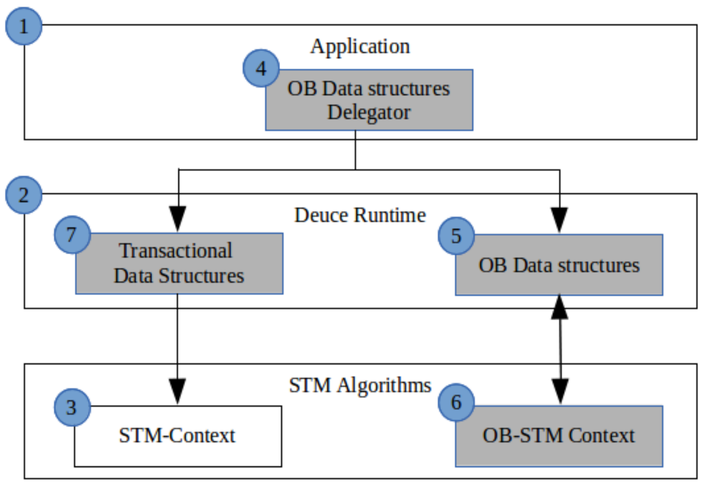


Figure 6.1: New design of DEUCE framework.

the context of a transaction. To achieve that, all methods (even if they are not *@Atomic*) are duplicated with an instrumented version, except those in an excluded class. Also, *@Atomic* methods are modified to the form of a retry loop calling the instrumented versions. Some optimizations are made to build these instrumented versions. More details about these optimizations can be found in [67].

STM context layer. STM context is an interface which allows programmers to extend the framework with more STM algorithms. DEUCE runtime interacts with STM algorithms using only this interface. Thus, the context interface includes the basic methods for any STM algorithm, like *init*, *commit*, *rollback*, *onReadAccess*, and *onWriteAccess*.

6.1.2 New Building Blocks to Support OTB

The design of our framework extension has three goals: *1)* keeping the simple programming interface of DEUCE; *2)* allowing programmers to integrate OTB data structures' operations with memory reads/writes in the same transaction; and *3)* giving developers a simple API to plug in their own OTB data structures and/or OTB-STM algorithms. To achieve that, we added the following four building blocks to DEUCE framework.

OTB Data Structures Delegator. In our new framework design, the application interface is extended with the ability of calling OTB data structures' operations. For example, in Algorithm 2, the user should be able to instantiate *set1*, and call its operations from outside the DEUCE runtime agent. At the same time, OTB data structures have to communicate with STM algorithms to guarantee consistency of the transaction as a whole. This means that OTB data structures have to interact with both the application layer and the DEUCE runtime layer.

To isolate the applications interface from the DEUCE runtime agent, we use two classes for each OTB data structure. The main class, which contains the logic of the data structure, exists in the runtime layer (inside the DEUCE agent). The other class exists at the application layer (outside the DEUCE agent), and it is just a delegator class which wraps calls to the internal class operations.

This way, the proposed extension in the applications interface does not affect programmability. There is no need for any addition to the language or any modifications in the JVM (like the original DEUCE interface). Also, all synchronization overheads are hidden from the programmer. The only addition is that the programmer should include delegator classes in his application code and call OTB operations through them.

OTB Data Structures. Calls from the application interface are of two types. The first type is traditional memory reads/writes, which are directly handled by the OTB-STM context (as described in the next block). The second type is OTB operations, which are handled by a new block added to DEUCE runtime, called OTB-DS (or OTB data structures). The design of an OTB data structure should satisfy the following three points:

- The semantics of the data structure should be preserved. For example, set operations should follow the same logic as if they are executed serially. This is usually guaranteed in optimistic boosting using a validation/commit procedure as shown in [48]. As a case study, in Section 6.2.1, we show in detail how the semantics of linked-list-based set are satisfied using such a validation/commit procedure.

- Communication between OTB-DS and OTB-STM algorithms. As shown in Figure 6.1, OTB data structures communicate with STM algorithms in both directions. On one hand, when an OTB operation is executed, it has to validate the previous memory accesses of the transaction, which requires calling routines inside the STM context. On the other hand, if a transaction executes memory reads and/or writes, it may need to validate the OTB operations previously called in the transaction.

- The logic of the underlying STM algorithm, which affects the way of interaction between OTB-DS and OTB-STM context. For example, as we will show in detail in Section 6.2, OTB-Set interacts with NOrec [25] and TL2 [31] in different ways. In the case of NOrec, which uses a global coarse-grained lock, acquiring semantic locks in OTB-DS may be useless because all transactions are synchronized using the global lock. On the contrary, TL2 uses a fine-grained locking mechanism, which requires OTB-DS to handle fine-grained semantic locks as well. It is worth noting that although a general way of interaction between OTB-DS and OTB-STM can be found, this generality may nullify some optimizations which are specific to each STM algorithm (and each data structure). We focus now on the specific optimizations that can be achieved separately on the two case-study STM algorithms (NOrec and TL2), and we keep the design of a general interaction methodology that works with all STM algorithms as a future work.

To satisfy all of the previous points, while providing a common interface, OTB-DS implements an interface of small sub-routines. These subroutines allow flexible integration between OTB operations and memory reads/writes.

- *preCommit:* which acquires any semantic locks before commit.
- *onCommit:* which commits writes saved in the semantic write-sets.
- *postCommit:* which releases semantic locks after commit.
- *validate-without-locks:* which validates semantics of the data structure without checking the semantic locks' status.
- *validate-with-locks:* which validates both the semantic locks and the semantics of the data structure.

Each OTB-STM context calls these subroutines inside its contexts in a different way, according to the logic of the STM algorithm itself. If a developer designs a new OTB-STM algorithm which needs a different way of interaction, he can extend this interface by adding new subroutines. It is worth noting that an *@Exclude* annotation is used for all OTB-DS classes to inform DEUCE runtime not to instrument their methods.

OTB-STM Context. As we showed in Section 6.1.1, STM context is the context in which each transaction will execute. OTB-STM context inherits the original DEUCE STM context to support OTB integration. We use a different context for OTB to preserve the validity of the applications which use the original DEUCE path (through block 7). OTB-STM context adds the following to the original STM context:

- An array of *attached* OTB data structures, which are references to the OTB-DS instances that have to be instrumented inside the transaction. Usually, an OTB data structure is attached when its first operation is called inside the transaction.
- Semantic read-sets and write-sets of each attached OTB data structure. As the context is the handler of the transaction, it has to include all thread local variables, like the semantic read-sets and write-sets.
- Some abstract subroutines which are used to communicate with the OTB-DS layer. In our case study described in Section 6.2, we only need two new subroutines: *attachSet*, which informs the OTB-STM context to consider the set for any further instrumentation, and *onOperationValidate*, which makes the appropriate validation (at both memory level and semantic level) when an OTB operation is called.

To implement a new OTB-STM algorithm (which is usually a new version of an already existing STM algorithm like NOrec and TL2, not a new STM algorithm from scratch), developers define an OTB-STM context for this algorithm and do the following:

- Modify the methods of the original STM algorithm to cope with the new OTB characteristics. Basically, *init*, *onReadAccess*, *commit*, and *rollback* are modified.
- Implement the new subroutines (*attachSet* and *onOperationValidate*) according to the logic of the STM algorithm.

Like OTB-DS, all OTB-STM contexts have to be annotated with *@Exclude* annotations.

Transactional Data Structures. This block is only used to support a unified application interface for both OTB-STM algorithms and traditional STM algorithms. If the programmer uses a traditional (non-OTB) STM algorithm and calls an OTB-DS operation inside the transaction, DEUCE runtime will use a traditional pure STM implementation of the data structure to handle this operation, and it will not use optimistic boosting anymore.

6.2 Case Study: Linked List-Based Set

Following the framework design in Section 6.1, we show a case study on how to integrate an optimistically boosted version of a linked-list-based set with the modified DEUCE framework[1]. This is done using the following two steps:

- Modifying OTB-Set implementation (described in Section 4.1) to use OTB-DS interface methods.
- Implementing OTB-STM algorithms which interact with the new OTB-Set. We use two algorithms in this case study, NOrec and TL2. As we showed in Section 6.1, we will need to implement a new OTB-STM context for both algorithms[2].

6.2.1 OTB-Set using OTB-DS interface methods

OTB-Set is communicating with the context of the underlying OTB-STM algorithm using the subroutines of the OTB-DS interface. OTB-Set implements these subroutines as follows:

Validation: Transactions validate that read-set entries are semantically valid. In addition, to maintain isolation, a transaction has to ensure that all nodes in its semantic read-set are not locked by another writing transaction during validation. As it is not always the case (in some cases, semantic locks are not validated, as shown in the next section), it is important to make two versions of validation:

[1]Skip-list-based OTB-Set is implemented in a similar way with few modifications.

[2]Note that the original STM contexts of NOrec and TL2 can still be used in our framework (using block 7 in Figure 6.1), but they will use an STM-based implementation of the set rather than our optimized OTB-Set.

- *validate-without-locks*: This method validates only the read-set and does not validate the semantic locks.
- *validate-with-locks*: This method validates both the values of the read-set and the semantic locks.

Commit: To be flexible when integrating with the STM contexts, commit consists of the following subroutines:

- *preCommit*: which acquires the necessary semantic locks on the write-sets. Like lazy linked-list: any `add` operation only needs to lock *pred*, while `remove` operations lock both *pred* and *curr*. This can be easily proven to guarantee consistency, as described in [54].
- *postCommit*: which releases the acquired semantic locks after commit.
- *onAbort*: which releases any acquired semantic locks not yet released when abort is called.
- *onCommit*: which publishes writes on the shared linked-list.

6.2.2 Integration with NOrec

To integrate NOrec with OTB-Set, two main observations have to be taken into consideration. First, using a single global lock to synchronize memory reads/writes can be exploited to remove the overhead of the fine-grained semantic locks as well. Semantic validation has to use the same global lock because in some cases the validation process includes both semantic operations and memory-level reads. As a result, there is no need to use any semantic locks given that the whole process is synchronized using the global lock. Second, both NOrec and OTB-Set use some kind of value-based validation. There are no timestamps attached with each memory block (like TL2 for example). This means that both NOrec and OTB-Set require an incremental validation to guarantee opacity [42]. They both do the incremental validation in a similar way, which makes the integration straightforward.

The implementation of OTB-NOrec context subroutines is as follows[3]:

init: In addition to clearing the memory-based read-set and write-set, each transaction should clear the semantic read-sets and write-sets of all previously attached OTB-Sets, and then it detaches all of these OTB-Sets to start a new empty transaction.

attachSet: This procedure is called in the beginning of each set operation (modifying Algorithm 3). It simply checks if the set is previously attached, and adds it to the local array of the attached sets if it is not yet attached.

[3]We skipped the implementation details of NOrec itself (and TL2 in the next section), and concentrate only on the modifications we made on the context to support OTB.

onOperationValidate: As both memory reads and semantic operations are synchronized and validated in the same way (using the global lock and a value based validation), this method executes the same procedure as *onReadAccess*, which loops until the global lock is not acquired by any transaction, and then it calls the *validate* subroutine.

validate: This private method is called on both *onReadAccess* and *onOperationValidate*. It simply validates the memory-based read-set as usual, and then validates the semantic read-sets of all the attached OTB-Sets. This validation is done using the *validate-without-locks* subroutine, which is described in Section 6.1.1, because there is no use of the semantic locks in OTB-NOrec context. If validation fails in any step, an abort exception is thrown.

commit: There is no need to call the attached OTB-Sets' *preCommit* and *postCommit* subroutines during transaction commit. Again, this is because these subroutines deal with semantic locks, which are useless here. The commit routine simply acquires the global lock, validates read-sets (both memory and semantic read-sets) using the *validate* subroutine, and then starts publishing the writes in the shared memory. After the transaction publishes the memory-based write-set, it calls the *onCommit* subroutine in all of the attached OTB-Sets, and then it releases the global lock.

rollback: Like *preCommit* and *postCommit*, there is no need to call OTB-Set's *onAbort* subroutine during the rollback.

6.2.3 Integration with TL2

TL2 [31], as opposed to NOrec, uses a fine-grained locking mechanism. Each memory block has a different lock. Reads and writes are synchronized by comparing these locks with a global version-clock. Also, unlike NOrec, validation after each read is not incremental. There is no need to validate the whole read-set after each read. Only the lock version of the currently read memory block is validated. The whole read-set is validated only at commit time and after acquiring all locks on the write-set.

Thus, the integration with OTB-Set requires validation and acquisition of the semantic locks in all steps. That is why we provide two versions of validation (with and without locks) in the layer of OTB-DS. The implementation of OTB-TL2 context subroutines is as follows:

init: It is extended in the same way as OTB-NOrec.

attachSet: It is implemented in the same way as OTB-NOrec.

onOperationValidate: There are two differences between OTB-NOrec and OTB-TL2 in the validation process. First, there is no need to validate the memory-based read-set when an OTB-Set operation is called. This is basically because TL2 does not use an incremental validation, and OTB-Set operations are independent from memory reads. Second, OTB-Sets should use the *validate-with-locks* subroutine instead of *validate-without-locks*, because semantic locks are acquired during commit.

onReadAccess: Like *onOperationValidate*, this subroutine has to call the *validation-with-locks* subroutines of all of the attached sets in addition to the original memory-based validation.

commit: Unlike OTB-NOrec, semantic locks have to be considered for the attached OTB-Sets. Thus, *preCommit* is called for of all the attached OTB-Sets right after acquiring the memory-based locks, so as to acquire the semantic locks as well. If *preCommit* of any set fails, an abort exception is thrown. During validation, OTB-TL2 context calls the *validate-with-locks* subroutine of the attached sets, instead of *validate-without-locks*. Finally, *postCommit* subroutines are called to release semantic locks.

rollback: Unlike OTB-NOrec, *onAbort* subroutines of the attached sets have to be called to release any semantic locks that are not yet released.

6.3 Evaluation

We now evaluate the performance of the modified framework using a set micro-benchmark. In each experiment, threads start execution with a warm up phase of 2 seconds, followed by an execution of 5 seconds, during which the throughput is measured. Each experiment was run five times and the arithmetic average is reported as the final result.

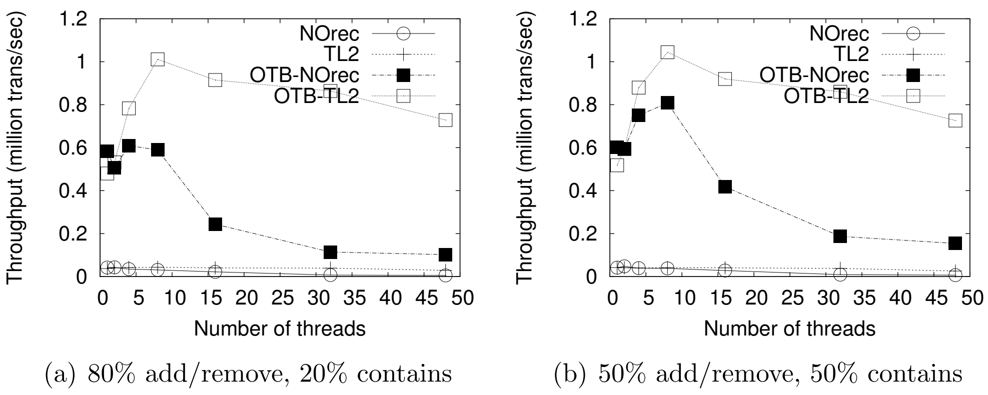
The experiments were conducted on a 48-core machine, which has four AMD Opteron (TM) Processors, each with 12 cores running at 1400 MHz, 32 GB of memory, and 16KB L1 data cache. The machine runs Ubuntu Linux 10.04 LTS 64-bit.

In each experiment, we compare the modified OTB-NOrec and OTB-TL2 algorithms (which are internally calling OTB-Set operations) with the traditional NOrec and TL2 algorithms (which internally call a pure STM version of the set).

We run two different benchmarks. The first one is the default set benchmark in DEUCE. In this benchmark, each set operation is executed in a transaction. This benchmark evaluates the gains from using OTB data structures instead of the pure STM versions. However, they do not test transactions which call both OTB operations and memory reads/writes. We developed another benchmark (which is a modified version of the previous one) to test such cases. In this second benchmark, as shown in Algorithm 2, each transaction calls an OTB-Set operation (add, remove, or contains), and increment some shared variables to calculate the number of successful and unsuccessful operations. As a result, both OTB-Set operations and increment statements are executed atomically. We justify the correctness of the transaction execution by comparing the calculated variables with the (non-transactionally calculated) results from DEUCE benchmark.

6.3.1 Linked-List Micro-Benchmark

Figure 6.2 shows the results for a linked-list with size 512. Both OTB-NOrec and OTB-TL2 show a significant improvement over their original algorithms, up to an order of magnitude of improvement. This is reasonable because pure STM-based linked-lists have a lot of false conflicts, as we described earlier. Avoiding false conflicts in OTB-Set is the main reason for this significant improvement. The gap is more clear in the single-thread case, as a consequence of the significant decrease in the instrumented (and hence logged) reads and writes. It is worth noting that in both versions (with and without OTB), TL2 scales better than NOrec, because NOrec is a conservative algorithm which serializes commit phases using a single lock.

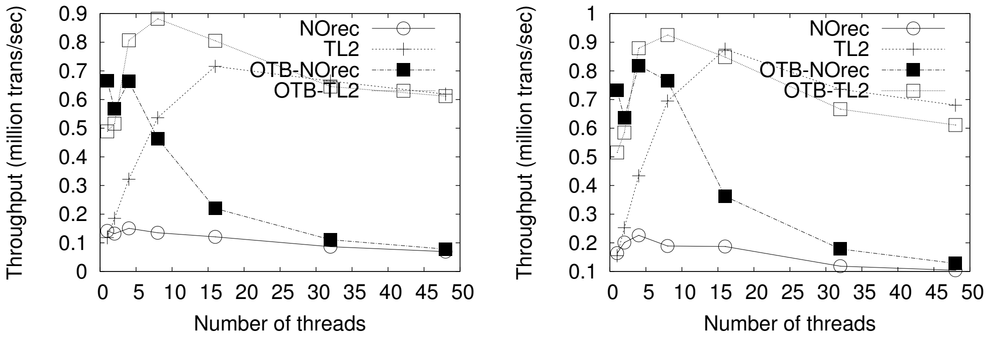


(a) 80% add/remove, 20% contains (b) 50% add/remove, 50% contains

Figure 6.2: Throughput of linked-list-based set with 512 elements, for two different workloads.

6.3.2 Skip-List Micro-Benchmark

Results for skip-list are shown in Figure 6.3. Skip-lists do not usually have the same number of false conflicts as linked-lists. This is because traversing a skip-list is logarithmic, and the probability of modifying the higher levels in a skip-list is very small. That is why the gap between OTB versions and the original STM versions is not as large as for linked-lists. However, OTB versions still perform better in general. OTB-NOrec is better in all cases, and it performs up to 5x better than NOrec for a small number of threads. OTB-TL2 is better than TL2 for a small number of threads, and it is almost the same (or slightly worse) for a high number of threads. Performance gain for a small number of threads is better because false conflicts still have an effect on the performance. For higher numbers of threads contention increases, which reduces the ratio of false conflicts compared to the real conflicts. This reduction in false conflicts reduces the impact of boosting, which increases the influence of the integration mechanism itself. That's why OTB-TL2 is slightly worse. However, plots in general show that the gain of saving false conflicts dominates this overhead in most cases.

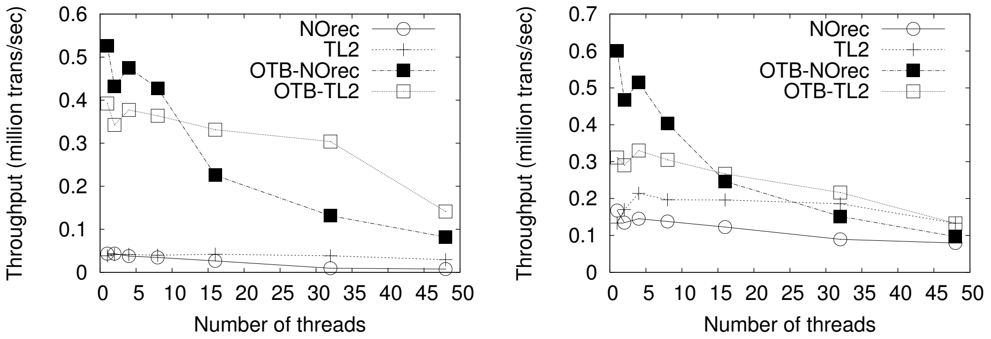


(a) 80% add/remove, 20% contains (b) 50% add/remove, 50% contains

Figure 6.3: Throughput of skip-list-based set with 4K elements, for two different workloads.

6.3.3 Integration Test Case

In this benchmark (Figure 6.4), we have six shared variables in addition to the shared OTB-Set (number of successful/unsuccessful adds, removes and contains). Each transaction executes a set operation (50% reads) and then it increments one of these six variables according to the type of the operation and its return value. As all transactions are now executing writes on few memory locations, contention increases and performance degrades on all algorithms. However, OTB-NOrec and OTB-TL2 still give better performance than their corresponding algorithms. The calculated numbers match the summaries of DEUCE, which justifies the correctness of the transactions. Also, NOrec versions relatively perform like the previous case (without increment statements), compared to TL2 versions. This is because NOrec (like all coarse-grained algorithms) works well when transactions are conflicting by nature.



(a) linked-list (b) skip-list

Figure 6.4: Throughput of Algorithm 2 (a test case for integrating OTB-Set operations with memory reads/writes). Set operations are 50% add/remove and 50% contains.

6.4 Summary

In this chapter we presented an extension of the DEUCE framework to support integration with transactional data structures that are implemented using the idea of *Optimistic Transactional Boosting*. As a case study, we implemented OTB-Set, an optimistically boosted linked-list-based set, and showed how it can be integrated in the modified framework. We then show how to adapt two different STM algorithms (NOrec and TL2) to support this integration. Performance of micro-benchmarks using the modified framework is improved by up to 10x over the original framework.

Chapter 7

Integrating OTB with HTM

The main issue of HTM processors is their *best-effort* nature, which means that transactions are not guaranteed to progress in HTM even in absence of conflicts, thus an efficient software fall-back path is still a mandatory requirement. Other issues are raised because of relying on the cache as a repository to keep the memory locations read and written by an ongoing transaction, such as the limited size of the cache, the false conflicts due to mapping different memory locations to the same cache line, and the possibility of aborting a transaction (especially if it is long) due to external interferences such as context switches or interrupts. The main side-effect of those issues is having transactions that are *prone to frequent aborts*.

The probability of having the aforementioned issues clearly increases when transactions include data structure operations, as we showed in Chapter 3. However, exploiting HTM in OTB is still feasible. One approach that has been already investigated in OTB-like approaches [9, 101] (which we named OSS in Chapter 3) is to execute only the *commit* phase as an HTM transaction. This way, both the footprint of the HTM transactions and the probability of having false conflicts are minimized. In this chapter, we provide the opposite: a methodology and a practical framework to inject efficient semantic operations (e.g., operations on a data structure with a certain semantics) into the commonly used *generic* HTM algorithms. (in this chapter, we use to term HTM also for hybrid transactions, i.e. the combination of fast-path and slow-path).

As we mentioned before, all HTM algorithms have two paths of execution: a *fast* HTM path, and a *slow* software fall-back path. The fall-back path is clearly slow because it either uses a global lock that serializes all the ongoing transactions, or uses a relatively slow STM algorithm such as NOrec [26, 88]. The fast-path, on the other hand, is not always as fast as it should be because it may repeatedly fail due to the HTM limitations mentioned before. Our methodology boosts the capabilities of the HTM algorithms by *i)* injecting the *efficient* semantic operations into the slow-path, and *ii)* using an adaptation mechanism to decide for each transaction which is the most effective path to start with. This approach makes significant performance improvements when data structure operations are dominating because, by relying on the first point, the slow-path will perform *faster* due to exploiting the efficient semantic versions of those operations, and, by relying on the second point, the fast-path can be bypassed when it is *slower* than the slow-path (e.g., when it repeatedly fails).

Clearly, semantic operations cannot be injected into the fast-path because the HTM APIs provide no control on that path (for example, Intel's TSX extensions simply speculate every memory access between the start and the end of the transaction). However, we claim that even if it is possible to inject semantics in the HTM path, it may not be the best alternative: if HTM is likely to succeed, then the best alternative would be to keep it as is; any trial to include semantics in this path will have a negative effect. On the other hand, if HTM is prone to failure, the same goal of modifying the fast-path is achieved by bypassing the fast-path to the *enhanced* slow-path in our proposal.

To inject semantic operations into the slow-path, we add two advanced versions of the conventional transactional read-set and write-set (we name them *abstract read-set* and *abstract write-set*). More in detail, any read-set entry can be seen as a generic entry equipped with one method, *isValid*, which is used to ensure that the read-set entry is still consistent with the other transactional reads and writes. Similarly, any write-set entry can be seen as a generic entry equipped with one method, called *writeback*, which is used to publish the entry into the shared memory at commit. We show that having those advanced read-sets and write-sets is enough to allow semantics in the slow-path of the common HTM algorithms.

From the practical perspective, we deployed our methodology by extending RSTM, a C++ STM framework [1]. We applied our proposal to a set of recent HTM algorithm [26, 88, 74] and we added them to RSTM (along with the original memory-based version of each). The design of our extended framework achieves three goals: *1)* keeping the simple programming interface of TM frameworks while supporting both HTM and STM; *2)* allowing programmers to integrate transactional data structure operations with memory reads/writes in the same transaction; and *3)* allowing developers to plug-in their own semantic operations through a simple API offered by the framework. The last two points enrich RSTM framework with the same semantics we already provided in DEUCE framework (in Chapter 6).

In conclusion, our framework allows HTM algorithms to work efficiently in different execution scenarios:

- In those scenarios favorable for HTM transactions, it works as efficient as the original HTM algorithm because its fast-path is not affected by the integration, which means that it maximizes the benefit of the underlying HTM support.
- In cases where HTM transactions likely fail, the adaptation phase allows an efficient shortcut to the slow-path.
- In cases where false conflicts are dominating (which inherently result in repeated failures in the fast-path as well), the slow-path exploits the integration with the semantic operations,

which results in much better performance.

- Also, if the underlying hardware does not support HTM, our framework can be used in a pure software mode by running all transactions in the slow-path from the beginning.

7.1 Can I Integrate My Transactional Data Structures?

In this section we show the requirements needed for a data structure to be integrated with our framework. The framework we propose can integrate data structures with any semantics, as long as their operations can be divided into the *traversal* and *update* phase. This separation is not enforced by the framework itself, whereas it is required that the data structure follows such a scheme. All the methodologies mentioned in Section 3.4 (including OTB) can be independently proven to be correct and compliant with that scheme. To move forward and allow other data structures to be integrated into our framework, we summarize the key points that the new implementations should satisfy in the following lemmas. In Chapter 10, we give more formal details about those points.

Lemma 1. *The traversal phase is always read-only.*

This phase is a pre-processing phase that calculates the return value of the operation and saves it to be validated later. All the actual modifications on the data structure should be done in the *update* phase.

Lemma 2. *Validation is always sound and pessimistic.*

No false positives are allowed. A validation may fail for a consistent scenario but it cannot succeed for inconsistent ones. The parameters passed to the validation procedure should be enough to detect any invalidation due to any concurrent operation. That is important to keep the data structure always consistent (i.e., after each operation and during commit).

Lemma 3. *All internal dependencies are solved during commit.*

If two dependent operations are executed in the same transaction (e.g., inserting and removing the same key), the result of the first operation should be propagated to the second one.

Given such a *correct* data structure, the following rules should also be satisfied by the design of the semantic operation to allow the integration with our framework.

- Data structures are accessed only using their APIs. The internal nodes of each data structure should never be accessed as a traditional memory locations.

- Each data structure operation should be cloned with a classical sequential version. That is because, if the transaction is in the fast-path mode, the operation should be executed inside an HTM transaction.

- The HTM (sequential) version should run safely with any concurrent (non-instrumented) *traversal* phase of a semantic operation on the same data structure (running in the slow-path). This rule requires moving any memory reclamation outside the HTM transaction. The problem of memory reclamation is orthogonal to our work and has been addressed before in different ways (e.g., [80, 6]).

- The output of the *traversal* phase is saved in the local abstract read/write sets (our framework supports any generic form of the sets' entries).

- All the logic used to validate and commit the operation is encapsulated inside two methods, named *isValid* and *writeback*. This abstraction allows the framework to validate the *traversal* phase at any time and to execute the *update* phase (which is encapsulated in the *writeback* method) during the commit of the enclosing transaction.

Theorem 1. *Data structures that respect those rules are decomposable.*

Proof. As each data structure is accessed only using its APIs, the validation of the *traversal* phase and the commit of the *update* phase of one data structure's operation is not affected by (and does not affect) the validation and the commit of any operation belongs to another data structure. □

7.2 HTM Algorithms

We now move to the *semantically-enabled* implementations of the known HTM algorithms. In the following three subsections, we discuss three common hybrid TM algorithms with different fall-back paths (HTM-GL, HyNOrec, and NOrecRH), and we show how it is possible to inject semantic operations into the fall-back path as long as they do not contradict the rules illustrated in Section 7.1. Then, in Section 7.2.4, we show how our methodology can be applied to other HTM algorithms.

7.2.1 Global Locking (HTM-GL)

The default fall-back mechanism for an HTM transaction is to protect the whole slow-path with a global lock[1]. The only requirement for safely executing both fast-path and slow-path

[1]In fact, TSX extensions allow this straightforward path using another mode (other than RTM) called *Hardware Lock Elision* (HLE). However, we stick with RTM to be able to modify the implementation of the fall-back path.

concurrently is to check the global lock at the very beginning of the fast-path; otherwise, pathologies like those illustrated in [32] can occur.

Algorithm 17 shows how we inject semantic operations into the slow-path (we call the new algorithm S-GL). First, the global lock is not acquired at the beginning of the fall-back path. Instead, it is acquired before the first *memory-based* read or write (or at commit if there is not any). If a semantic operation appears before the first memory-based read/write, the transaction executes *only* the *traversal* phase of the operation and saves the results in the corresponding abstract read-set (and the abstract write-set, if the operation is writing), then the whole abstract read-set is validated (each entry is validated using its own *isValid* method) to ensure that the semantic operations executed so far are all still consistent.

This way, we allow the execution of as many *traversal* phases as possible, performed concurrently with other transactions' fast-paths and with other *traversal* phases, before acquiring the global lock. Once the slow-path reaches either the first memory read/write or the commit phase, it acquires the global lock, then validates any existing semantic operation in the abstract read-set (using the per-entry *isValid* method), and completes the *update* phase of any semantic operation in the abstract write-set (using the per-entry *writeback* method).

From this point on, the execution will be similar to the original implementation. Clearly, any semantic operation that appears after the first memory-based read/write will be executed in the traditional sequential mode because there is no need to add more overhead after the global lock acquisition.

S-GL pays off only if the semantic operations appear early in the transaction. However, we discovered that this situation is a common trend in several benchmarks and real applications (e.g., some transactions in Genome execute only semantic operations without any memory read/write). Also, in the worst case when the transaction starts by a memory operation, S-GL performs as the original HTM-GL algorithm. This issue will be solved by the following algorithms in Section 7.2.2.

Theorem 2. *HTM transactions that use S-GL as a fall-back path are consistent.*

Proof. The correctness of our approach is easy to show. In the slow-path, as long as the operations respect the rules mentioned in Section 7.1, all the traversal phases executed before the first memory access are consistent because each operation is followed by a validation of the whole abstract read-set. Once the global lock is acquired, it is guaranteed that no other concurrent transaction is currently in its fast-path or in the commit phase of its slow-path. At this point, the abstract read-set is validated again and the abstract write-set is published, then the execution is entirely protected by the global lock. Although deferring the lock acquisition has been known to have side effects on the correctness of the live transactions [32], those side effects cannot happen in our case, because we prevent any memory access before the lock acquisition and we allow only the *traversal* phase of the semantic operations to be executed. Those *traversal* phases are consistent by relying on the correctness of the underlying data structure. □

Algorithm 17 S-GL Algorithm.

```
procedure FAST-PATH-POST-BEGIN
    if isLocked(lock) then
        TM-ABORT
end procedure

procedure SLOW-PATH BEGIN
    // Do not acquire the lock here.
end procedure

procedure SLOW-PATH onFIRSTMEMORYACCESS
    commit-pending-semantics()
end procedure

procedure SLOW-PATH COMMIT
    commit-pending-semantics()
    release(lock)
end procedure

procedure SEMANTIC-OPERATION
    if Fast-Path OR afterFirstMemoryAccess then
        Execute the sequential version.
    else
        Execute ONLY the traversal phase.
        Save the result in the abstract read(write)-set.
        Validate the abstract read-set.
end procedure

procedure COMMIT-PENDING-SEMANTICS
    acquire(lock)
    if isValid(abstract-read-set) = false then
        release(lock)
        TM-ABORT
    writeback(abstract-write-set)
end procedure
```

7.2.2 Hybrid NOrec (HyNOrec)

Two hybrid TM algorithms [26, 88] propose NOrec as the fall-back software path of HTM transactions. The main reason of selecting NOrec is that it has minimal meta-data (only one global lock acquired at commit time), which means that HTM transactions need only to check (and update) this global lock in order to avoid any conflict with STM transactions. In those proposals, to safely fall back to NOrec, two modifications are made on the fast-path: the global lock is checked at the beginning of the transaction (like HTM-GL), and the global lock's version is incremented at the end of the transaction. The latter requirement is important because the fall-back path now is not completely protected by the global lock (this lock is acquired only at commit) and the slow-path should be notified somehow about the newly-committed fast-paths. Although this increment generates conflicts among fast-paths, its effect is minimal because it is done at the very end of the HTM transaction.

In this section we enrich HyNOrec with semantic operations, thus producing a new algorithm (named S-HyNOrec), as shown in Algorithm 18. Unlike S-GL, S-HyNOrec injects semantic operations into the slow-path along with memory-based reads/writes without the need of acquiring the global lock before the commit phase. If a read or write operation is executed, it is handled in the same way as the original HyNOrec algorithm, and saved in the original read-set or write-set, respectively. If a semantic operation is called, only its *traversal* part is executed and the needed information is saved in the semantic read-set or write-set. The validation and commit procedures in S-HyNOrec are the same as HyNOrec, except that they scan both the semantic and the memory-based read-set and write-sets (the semantic read-set and write-set are accessed using the *isValid* and *writeback* methods). To guarantee opacity [42], both the memory-based read-set and the abstract read-set are validated after each semantic operation, after each new memory read, and at commit phase.

To prove the correctness of S-HyNOrec, we first note that in the slow-path we may have both memory accesses and semantic operations coexisting in the read-set and the write-set. However, the consistency of each type is independent from the other because, as the rules in Section 7.1 state, the data structures are accessed only using their APIs. Corollary 1 shows that.

Corollary 1. *The consistency of a data structure as defined in Section 7.1 cannot be affected by (or affect) the other non-semantic memory accesses performed in the same transaction or in another concurrent transaction.*

Theorem 3. *HTM transactions that use S-HyNOrec as a fall-back path are consistent.*

Proof. The slow-path of S-HyNOrec follows the same scheme as the one of HyNOrec. The main difference is that S-HyNOrec has two different types of read-set and write-set. However, according to Corollary 1, those two types of sets are completely independent. As long as both types are considered during the transaction validation and commit, consistency is preserved.

Algorithm 18 S-HyNOrec Algorithm.

```
...
procedure Fast-Path-Post-Begin
    if isLocked(lock) then
        TM-ABORT
end procedure

procedure Fast-Path-Pre-Commit
    lock += 2 // The same as HyNOrec
end procedure

procedure Slow-Path Validate
    // Do not acquire the lock here.
end procedure

procedure Slow-Path onFirstMemoryAccess
    ...
    if isValid(memory-read-set) = false then
        TM-ABORT
    if isValid(abstract-read-set) = false then
        TM-ABORT
    ...
end procedure

procedure Slow-Path Commit
    ...
    writeback(memory-write-set)
    writeback(abstract-write-set)
    ...
end procedure

procedure Semantic-Operation
    if Fast-Path then
        Execute the sequential version.
    else
        Execute ONLY the traversal phase.
        Save the result in the abstract read(write)-set.
        Validate both the memory and the abstract read-sets.
end procedure
```

Live transactions are kept consistent because their read-sets are validated after each semantic operation, as well as after each non-semantic memory read. □

7.2.3 Reduced Hardware NOrec (NOrecRH)

Both HTM-GL and HyNOrec expose two main issues. First, the global lock has to be checked (and accordingly instrumented) at the beginning of the HTM transaction, which introduces several false conflicts. Second, the fall-back path uses either mutual exclusion or a pure STM algorithm, which is notably slower than the faster HTM path.

Reduced Hardware NOrec (NOrecRH) [74] has been proposed to solve those two issues. It uses an intermediate path between the HTM transaction and the pure STM execution. In this intermediate path, the body of the transaction is executed in a similar way as NOrec, but during commit an HTM transaction is used to apply the entries of the write-set to the shared memory instead of acquiring the global lock. The intermediate path is considered the slow-path, and the final (pure STM) path is called slow-slow-path.

In more details, the commit phase of the slow-path starts by reading the global lock in a local snapshot, re-validating the read-set, and then using a short HTM transaction as explained above. Inside this small HTM transaction, the version of the lock is re-checked to make sure that it is not changed during the writing process. This way, both the fast-path and the slow-path read and increment the global lock only at the end of the HTM transaction (because all conflicts between them will be solved by the HTM hardware conflict detection itself). Doing so decreases the window of false invalidation due to the conflict on the global lock. Additionally, transactions are now more likely to succeed in the slow-path, even if the main transaction body contains external causes of HTM aborts (e.g., system calls and page faults) because the small HTM transaction executed during the slow-path's commit phase is only responsible for publishing the write-set.

The slow-slow-path is synchronized with the other paths by acquiring another lock, which is instrumented at the beginning of the HTM transactions running in the fast-path and the slow-path. Although the overhead of the two former HyNOrec proposals seems the same, the likelihood of falling back to the slow-slow-path in NOrecRH is much lower because transactions try (and likely succeed) in the slow-path first.

Semantic operations are injected into NOrecRH as we explained in Section 7.2.2 for HyNOrec. The only difference is that semantic operations are now injected into both the slow-path and the slow-slow-path. In the slow-path, the execution of the *update* phase is atomically executed (along with any other memory write) using the enclosing short HTM transaction, which guarantees consistency against the fast-path. In the slow-slow-path, the (second) global lock protects the execution of the *update* phase and avoids conflicting with any other semantic operation executed in any of the three paths.

One distinguishing point between the original NOrecRH and S-NOrecRH is in the decision of when to fall back to the slow-slow-path. The original NOrecRH assumes that the small HTM transaction (executed at the end of the slow-path) cannot abort for any external reason, like a system call invocation. That is because the small transaction is in charge of just publishing the values stored in the transaction's write-set into the shared memory. For this reason, they only fall back to the slow-slow-path if the failure is for capacity. This assumption is acceptable in absence of the semantic operations, because failures due to conflicts can be eventually solved, and failures due to external sources cannot happen. However, this assumption does not hold anymore with S-NOrecRH, because semantic operations are now integrated and the *writeback* method depends on the semantic of each data structure. As an example, if the write-set entry represents an insertion into a linked-list, it will need to allocate memory for the new node, which may result in repeated page faults (resulting in a failure with *_xabort_other* flag). For that reason, we experienced infinite loops in our experiments when we adopted the same assumption as NOrecRH. To overcome this problem, S-NOrecRH falls back to the slow-slow-path if the HTM failure is, for any reason, different from $\_xabort\_conflict$.

Theorem 4. *HTM transactions that use S-NOrecRH as a fall-back path are consistent.*

Proof. Leveraging the Corollary 1, the proof follows the one of S-HyNOrec. □

7.2.4 Other Hybrid Algorithms

Our methodology is not limited to the above algorithms. We can infer that from Theorem 1 and Corollary 1. Although both semantic and memory-based operations are allowed in the same transaction, they are physically isolated through the read-set and the write-set, and they are validated and committed independently from each other. That is why injecting semantic operations into the slow-path, which is mandatory in any hybrid TM algorithm, is always possible. We list below some other hybrid TM algorithms that can also be extended according to our methodology:

- *Algorithms with lazy subscription.* A non-opaque [42] version of HTM-GL has been proposed in [18]. In this versions, the global lock is read at the end of the fast-path instead of its beginning (this technique is also called *lazy subscription*). Committed transactions are not affected by this modification; however, live transactions (that will eventually abort) may see an inconsistent snapshot of the memory because the changes by the concurrent slow-paths will be detected only at the end of the transaction [32]. The same idea of lazy subscription has been proposed as an enhancement of one of HyNOrec's two versions [88]. Our methodology can be applied to those versions because the slow-path is not changed.

- *TL2-RH.* Before NOrecRH, TL2 [31] has been proposed as a fall-back STM algorithm in the slow-path and the slow-slow-path [74]. However, TL2 is a fine-grained algorithm which has more meta-data than NOrec. That is why the performance of the fast-path (which has to instrument those meta-data) is significantly affected and the algorithm itself becomes much more complex. Although practically NOrecRH dominated TL2-RH in literature, our methodology is still compatible with the latter.

- *Invyswell:* The main idea of Invyswell [17] is to fall back to InvalSTM [40] instead of NOrec. In InvalSTM, the committing transaction uses bloom filters to detect any conflict with the live transactions and invalidate them accordingly. Applying our methodology on Invyswell is not straightforward like the others because semantic operations cannot be easily compacted into bloom filters. One way to solve this issue is to handle the semantic operations in a similar way to S-HyNOrec, and use the bloom filters (and the whole Invyswell mechanism) to invalidate the memory-based parts only.

7.3 The Framework

In this section we integrate the previously discussed *semantically enhanced* hybrid TM algorithms (see Section 7.2) into the RSTM framework. Then we show how these algorithms can be further enhanced by an adaptation mechanism that selects at runtime the best path to start with (see Section 7.4).

The design of our framework should maintain the simple API of TM and allow the injection of the efficient semantic operations. To achieve both goals, we extend RSTM in three steps, as shown in Figure 7.1 (gray blocks are the added/modified blocks). First, we add both the memory-based and the semantic-based versions of three Hybrid TM algorithms (HTM-GL, HyNOrec, and NOrecRH). Second, we add an abstract method, called `Semantic-Operation`, to the runtime layer of the framework, which enables the integration of any semantic operation with our framework as long as it follows the guidelines presented in Section 7.1. Finally, in the application interface layer, we modify the basic API constructs, like `TM-BEGIN`, `TM-END`, `TM-READ`, and `TM-WRITE`[2] to support both HTM and STM transactions, and we add a new API method, called `TM-OPERATION`, which allows the programmer to call any operation that extends `Semantic-Operation` inside a transaction. This abstraction isolates any low-level details, including the integration with the semantic operations and the exploitation of the HTM constructs, from the programmer. It is important to mention that any software designed for running in STM can still run in our modified RSTM framework without any modification and without suffering from any additional overhead.

Unified API

As shown in Figure 7.1, one of our goals is to provide a unified API to interact with the framework. Programmers should be isolated from all framework implementation decisions

[2]Unlike DEUCE, RSTM allows the programmer to identify which reads and writes in the transactional should be transactional (using the `TM-READ` and `TM-WRITE` primitives).

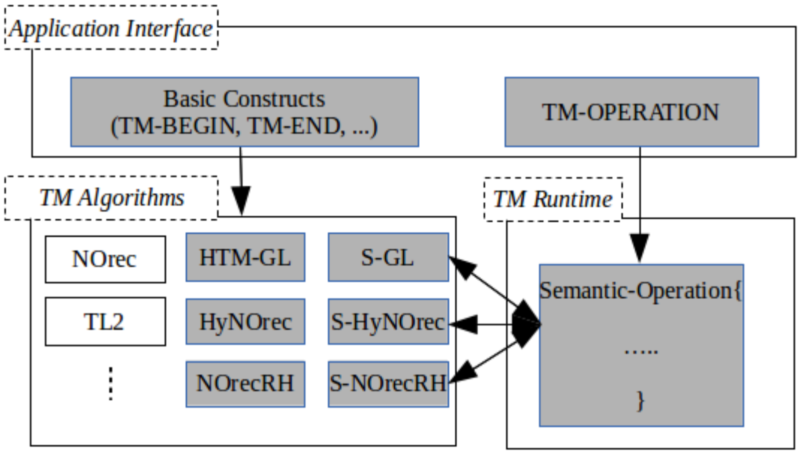


Figure 7.1: Extension of the RSTM framework.

such as: how to integrate data structures; checking the compatibility with HTM; and deciding in which path transactions should start or retry. To achieve that, our framework adds only one new API (called `TM-OPERATION`). All the other basic transactional APIs like `TM-BEGIN`, `TM-END`, `TM-READ`, and `TM-WRITE` are unchanged, whereas their implementation is slightly modified to implicitly decide in which path a transaction should execute without involving the programmer in this decision. For all the hybrid algorithms, transactions try five times in each path before falling back to the slower path. These modifications are reported in Algorithm 19.

For `TM-BEGIN`, if the used TM algorithm is a hybrid one, then the transaction starts first in the fast-path. Otherwise it starts directly with the slow-path. Doing so allows the framework to use the classical STM algorithms (e.g., NOrec and TL2) as usual. Additionally, if the underlying hardware has no HTM support, the framework automatically switches from the hybrid algorithm to the pure STM versions of it (the default is NOrec). The other basic constructs (`TM-READ`, `TM-WRITE`, and `TM-END`) check if the transaction is in the fast-path or in another path and call the appropriate handling method accordingly.

The new `TM-OPERATION` API calls the methods that extend the internal `Semantic-Operation` interface, which is handled in each hybrid TM algorithm as mentioned in Section 7.2. This way, the programmer uses the same interfaces provided by RSTM, without the need to modify the previous benchmarks, and with the additional capability of calling any `TM-OPERATION` inside the transaction.

Algorithm 19 API Interfaces

```
 1: procedure TM-BEGIN
 2:     if No HTM Support OR STM Algorithm then
 3:         tx.path = SLOW-PATH
 4:     else
 5:         tx.path = FAST-PATH

 6:     // According to tx.path, use htm-begin or stm-begin
 7:     if tx.path = FAST-PATH then
 8:         tries = 5
 9:         while 1 do
10:             status = _xbegin
11:             if status = _xbegin_succeeded then
12:                 // Start HTM. Post-Begin differs according to the HTM algorithm
13:                 Fast-Path-Post-Begin
14:                 ...
15:                 break
16:             tries - -
17:             if tries <= 0 then
18:                 tx.path = SLOW-PATH
19:                 break
20:     // Fall-back slow-path
21:     if tx.path = SLOW-PATH then
22:         stm-begin()
23: end procedure

24: // All other constructs like TM-READ, TM-WRITE are the same
25: procedure TM-END
26:     if tx.path = FAST-PATH then
27:         // Finish HTM according to the HTM algorithm used
28:         Fast-Path-Pre-Commit
29:         _xend
30:     else
31:         // If NOrecRH or S-NOrecRH, they will implicitly decide if
32:         // stm-end will commit in slow-path or slow-slow-path
33:         stm-end()
34: end procedure
```

7.4 Adapting the Starting Path of the Hybrid Algorithms

Due to the limited API given by Intel's TSX extensions, the fast-path of any hybrid TM algorithm, which is completely executed in HTM, cannot exploit the semantic operations to reduce the false conflicts and the transaction's footprint. This limitation means that if the long traversals of the data structures represent the dominating overhead of the workload, the hybrid algorithms cannot avoid the overhead of the fast-path (that will likely fail).

To address this issue, we provide an adapted version of each of the three hybrid TM algorithms. In these versions, we use a simple heuristic to decide whether it is better to start with the fast-path or not. The metric we used in our technique is the overall percentage of failures in the fast-path for each thread. If the number of failures in the fast-path (after consuming the 5 trials) is larger than the number of successes, we consider that as an indication that the workload does not represent the best candidate for HTM transactions (for any of the aforementioned reasons: false conflicts; capacity limitations; and external sources). When such a situation occurs, we decide to execute the next n transactions directly in the slow-path, where n is the difference between the number of successful and failed transactions in the fast-path. Then, we resume the execution in the fast-path and re-calculate the metric. This simple mechanism is similar to the hill climbing mechanism with only two states (i.e., executing in the fast-path and executing in the slow-path).

Algorithm 20 Modified TM-BEGIN and TM-END for adapting the starting path.

```
1: procedure TM-BEGIN
2:     if No HTM Support OR STM Algorithm then
3:         $tx.path$ = SLOW-PATH
4:     else if $tx.slow\_path\_trials > 0$ then
5:         $tx.path$ = SLOW-PATH
6:     else
7:         $tx.path$ = FAST-PATH
8:     ...
9:     // Here is the point when a trial in the fast-path fails
10:    if tries <= 0 then
11:        $tx.path$ = SLOW-PATH
12:        $tx.fail\_count$++
13:        $tx.slow\_path\_trials = tx.fail\_count - tx.success\_count$
14:        break
15:    ...
16: end procedure

17: procedure TM-END
18:    if $tx.path$ = FAST-PATH then
19:        ...
20:        $\_xend$ // Successful fast-path
21:        $tx.success\_count$++
22:        $tx.slow\_path\_trials = tx.fail\_count - tx.success\_count$
23:    ...
24: end procedure
```

Algorithm 20 (which extends Algorithm 19) shows how we modified the `TM-BEGIN` and

`TM-END` APIs to implement this adaptive mechanism. After the transaction succeeds (fails) in the fast-path, it increases the local variables *tx.success_count* (*tx.fail_count*). Then, it updates another local variable, *tx.slow_path_trials*, which is the difference between those two variables. At the beginning of TM-BEGIN, if *tx.slow_path_trials* is greater than 0, then the transaction starts in the slow-path instead of the fast-path, and then it decreases *tx.slow_path_trials*. Thus, after each fast-path trial, n transactions will execute in the slow-path (if n is positive).

In the normal cases, in which the fast-path likely succeeds, n will remain negative during the whole execution, and will not affect the performance. Furthermore, all those variables are local to each transaction, so that they cannot produce any new conflict; thus they have a minimal effect on the execution.

In NOrecRH, we can also apply the same adaptation from the slow-path to the slow-slow-path. However, the likelihood of falling back to the slow-slow-path is very small due to the reasons we mentioned in Section 7.2.3. In fact, this path is added for safety issues rather than to enhance the performance. That is why we decided not to apply the adaptation to that phase.

Although we propose this adaptation phase mainly to enhance the semantic operations called within transactions, the same enhancement will be achieved if the transactions are expected to repeatedly fail for any other reason.

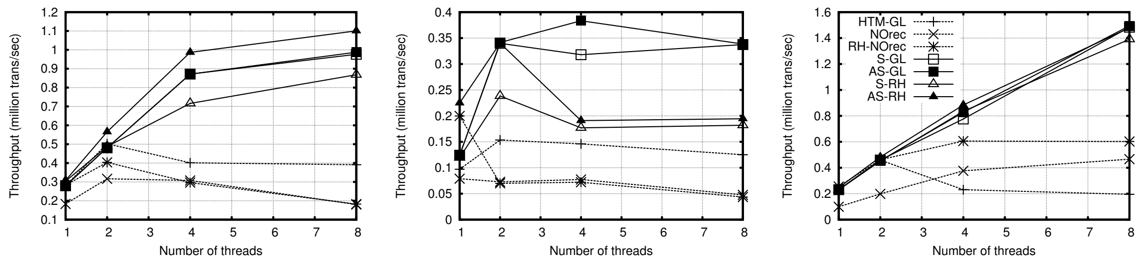
7.5 Evaluation

We developed our framework in C++ and integrated it with RSTM [1]. In this evaluation we compare the performance of transactions that invoke semantic operations (on OTB-Set detailed in Section 4.1) using S-GL and S-NOrecRH against transactions that use the traditional TM-based implementations of the linked-list and the skip-list. Our selection of the competitors includes: the default HTM-GL, as the memory based version of S-GL; NOrecRH, the memory based version of S-NOrecRH; and NOrec[25], a pure STM competitor. We also show the results of two versions of each semantic-based algorithm, with and without the adaptation process (named S-GL/S-RH and AS-GL/AS-RH, respectively). We ran each experiment 5 times and plotted the average of their results.

In all experiments, we use an Intel Haswell chip with 4 cores and 8 hardware threads (using hyper-threading). We first test the integration of the linked-list in Section 7.5.1, and then we show the performance for the skip-list in Section 7.5.2. Finally in Section 7.5.3, in order to assess the performance of our framework on a realistic workload, we show the performance using Genome, one of the applications belonging to the STAMP [81] benchmark. We selected Genome because it also executes transactions including memory and data structure operations.

7.5.1 Linked-List

In these experiments, we use a benchmark that executes five linked-list operations in each transaction. Additionally, if the operation is a successful (add or remove) operation, a shared counter is incremented inside the transaction to calculate (transactionally) the total number of the successful operations. This way, the workload contains both memory accesses and semantic accesses in the same transaction. The results with different sizes of the linked-list and different percentages of read operations are plotted in Figure 7.2. The object accessed by the operations are selected in a way such that the size of the linked-list is almost constant (i.e., no unbalancing).



(a) 256 nodes, 20% contains (b) 1024 nodes, 20% contains (c) 1024 nodes, 100% contains

Figure 7.2: Throughput of both the semantic-based and the memory-based TM algorithms in a linked-list benchmark with 5 operations/transaction (higher is better).

Figure 7.2(a) shows the results when 20% of the operations are `contains` operations, and the size of the linked-list is 256. The remaining 80% operations are not all write operations because some of them are expected to be unsuccessful `add` or `remove` operations. In this workload, the unmonitored traversal of semantic-based algorithms significantly reduces the signature of the transactions (i.e., the size of their read-set and write-set) in the slow-path, and in the slow-slow-path for S-NOrecRH. For this reason, the gap between our algorithms and the others increases, as they suffer from a higher probability of conflict due to the very long read and write signature. The maximum speedup observed between our semantic algorithms and the competitors is more than 3×.

In Figure 7.2(b), we repeated the same experiment after increasing the size of the linked-list up to 1024 elements. Here, because of the higher footprint of the transactions, the probability of failing in the fast-path of the HTM algorithms increases, along with the probability of having false conflicts due to the longer size of the list. As a result, the benefit of adopting our algorithm becomes more evident. Recall that if a `contain` operation accesses a node at the end of the linked-list, any concurrent insertions involving any of the previous nodes will raise a false conflict. Starting from 2 threads, our algorithms perform better than all competitors, and each algorithm achieves up to 4× better performance than its corresponding memory-based competitor. In this experiment, global locking algorithms (HTM-GL and S-GL) generally perform better than the corresponding NOrecRH algorithms (NOrecRH and

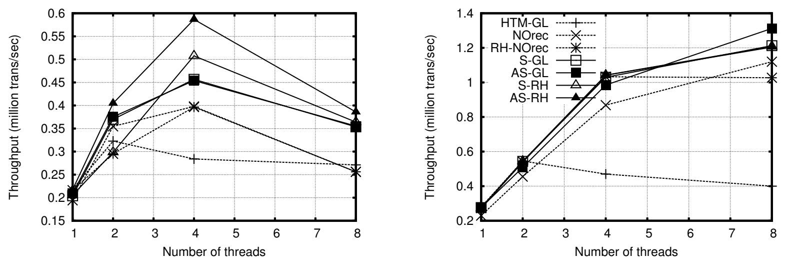
S-NOrecRH) for 4 and 8 threads because the contention increases and it becomes wise to serialize the slow-paths.

Figure 7.2(c) represents an important case that we made on purpose to show the performance of our framework under an adverse workload (i.e., without any false conflict). We did that by converting all operations to be `contains` operations (read-only scenario). It is important to note that, despite the absence of false conflicts on the linked-list, transactions in the NOrecRH approaches can still fail in HTM because they all have to increment the global lock at the end of the fast-path. Also, as specified by Intel's architects, HTM transactions can fail for any other external reason, like interrupts or OS scheduling. Anyway, in this case the probability of failing in the fast-path is lower than the case in Figure 7.2(b), and that is the reason why all algorithms scale better. For the cases in which the fast-path fails, our semantic algorithms perform better than the memory-based algorithms because they traverse the list (in the slow-path) without instrumentation. In this experiment, the adaptation enhancement does not have the same impact as in the other (writing) cases, because here conflicts are rare and the probability of failing while executing the fast-path is relatively small. Accordingly, HTM-GL is the worst because if a transaction fails for any reason, it suffers from a fruitless serialization because all operations are commutable (i.e., all contains).

One additional comment can be inferred from the analysis of Figure 7.2. The adaptation process always pays off. We did not discover any scenario where it affects the performance negatively. That is mainly because the adaptation does not impact the case in which transactions succeed in HTM, and, at the same time, it makes an effective shortcut to the slow-path if the fast-path likely fails. Also computing the heuristic has minimal overhead.

7.5.2 Skip-List

In these experiments, we assess the performance of our framework for the skip-list benchmark. As before, the size of the skip-list is kept stable.



(a) 20% contains (b) 100% contains

Figure 7.3: Throughput of both the semantic-based and the memory-based TM algorithms in a skip-list benchmark with 64K nodes and 5 operations/transaction (higher is better).

Figure 7.3(a) shows the results when 20% of the operations are `contains` operations and the size of the skip-list is 64K. Unlike the linked-list case, here TM-based algorithms scale better because the skip-list has a lower footprint and lower conflict probability (as it traverses the set in a logarithmic time using the higher levels of the skip-list). However, the semantic algorithms remain better because they still reduce the false conflicts. Unlike the linked-list, global locking algorithms generally perform worse than the corresponding NOrecRH algorithms for 4 and 8 threads because the contention in the skip-lists is lower. The adaptation process still pays off and enhances the performance of S-NOrecRH.

Figure 7.3(b) shows the case of the read-only workload. The trends are similar as those in Figure 7.2(c) of the linked-list. However, here the gap between NOrecRH and its semantic versions is smaller.

7.5.3 Genome

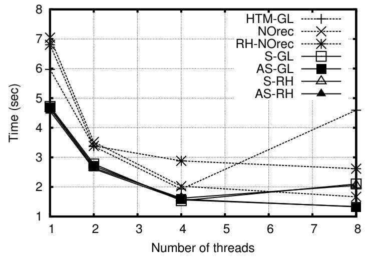


Figure 7.4: Execution time in genome benchmark (lower is better).

In these experiments, we assess the performance of our framework on Genome, one application of the STAMP benchmark that internally leverages a hash-table of linked-lists. The latter has the same interfaces as OTB-Set (presented in Section 4.1). Figure 7.4 plots the results. In Genome, some transactions call N insertions into the linked-lists, which may consist of hundreds of elements after some processing time. That is why the semantic-based algorithms always perform better than the memory-based algorithms. The only exception is for 8 threads, in which NOrec performs better than the non-adapted version of the semantic-based algorithms. Although negative, it is reasonable because the contention is high and the HTM fast-path repeatedly fails. However, the adapted version of the proposed algorithms outperforms NOrec as it promptly detects that it is better to start immediately from the slow-path.

7.6 Summary

In this chapter we introduced a methodology to include semantic operations alongside memory accesses within the same HTM transaction. We achieved that by injecting the semantic operations into the fall-back path of well-known HTM algorithms. Furthermore, we proposed an adaptation process that starts the transactions using the best execution path according to their observed conflicts. Our experiments show a better performance than the TM algorithms that access the data structure without exploiting its semantics.

Chapter 8

Remote Transaction Commit

In this chapter we present Remote Transaction Commit (RTC), a mechanism for processing commit phases of STM transactions remotely. RTC's basic idea is to execute the commit part of a transaction in dedicated servicing threads. In most STM algorithms, the commit part has high synchronization overhead, compared to the total transaction overhead (see Section 8.1.2 for a detailed discussion on this). Moreover, this overhead becomes dominant in high core count architectures, where the number of concurrent transactions can (potentially) increase significantly. By dedicating threads for servicing the commit phase, RTC minimizes this overhead and improves performance.

In RTC, when client transactions[1] reach the commit phase, they send a commit request to a server (potentially more than one). A transaction's read-set and write-set are passed as parameters of the commit request to enable the server to execute the commit operation on behalf of the clients. Instead of competing on spin locks, the servicing threads communicate with client threads through a cache-aligned requests array. This approach therefore reduces cache misses (which are often due to spinning on locks), and reduces the number of CAS operations during commit[2]. Additionally, dedicating CPU cores for servers reduces the probability of interleaving the execution of different tasks on those cores due to OS scheduling. Blocking the execution of commit phase, for allowing other transactions to interleave their processing on the same core, is potentially disruptive for achieving high performance.

RTC follows similar directions of lazy, lightweight, coarse-grained STM algorithms, like NOrec [25]. This way, it minimizes the number of locks that will be replaced by remote execution (precisely, it replaces only one lock). This also makes RTC privatization-safe [97], and a good candidate for hybrid transactional memory approaches [91]. Validation in RTC is value-based, like NOrec, which reduces false conflicts, and efficiently deals with non-transactional code. Moreover, RTC solves NOrec's problem of serializing independent

[1]We will call an RTC servicing thread as "server" and an application thread as "client".

[2]It is well understood that spinning on locks and increased the usage of CAS operations can seriously hamper application performance [59], especially in multicore architectures.

commit phases (because of the single lock) by using an additional secondary server to execute independent commit requests (which do not conflict with transactions currently committed on the main server). RTC exploits bloom filters to detect these independent requests. In Section 8.2, we show RTC's implementation using one main server and one secondary server. Then, in Section 8.6, we enhance RTC with more than one secondary server, and evaluate the impact of adding more servers on RTC's performance.

Extending more fine-grained algorithms like RingSTM and TL2 with remote commit is not as efficient as doing so with NOrec. With their higher number of locks, more complex mechanisms are needed to convert those locks into remote commit execution, which also requires more overhead for synchronization between servers. Although we use bloom filters to detect independent transactions, like RingSTM, we still use one global lock (not bloom filters) to synchronize transactions, as we will show later.

RTC is designed for systems deployed on high core count multicore architectures where reserving few cores for executing those portions of transactions does not significantly impact the overall system's concurrency level. However, the cost of dedicating cores on some architecture may be too high given the limited parallelism. For those architectures, we extend RTC by allowing application threads, rather than a dedicated server thread, to combine the execution of the commit phases. This idea is similar to the *flat combining* [55] approach proposed for concurrent data structures (in fact, we name the new version of RTC as RTC-FC). Our experiments include the comparison between RTC and RTC-FC, thus clarifying the workloads that can benefit more from one approach or the other.

Our implementation and experimental evaluation show that RTC is particularly effective when transactions are long and contention is high. If the write-sets of transactions are long, transactions that are executing their commit phases will be a bottleneck. All other spinning transactions will suffer from blocking, cache misses, and unnecessary *CASing*, which are significantly minimized in RTC. In addition, our experiments show that when the number of threads exceeds the number of cores, RTC performs and scales significantly better. This is because, RTC solves the problem of blocking lock holders by an adverse OS scheduler, which causes chained blocking.

Through experimentation, we show that both RTC and RTC-FC have low overhead, peak performance for long running transactions, and significantly improved performance for high number of threads (up to 4x better than other algorithms in high thread count configurations). We also show the impact of increasing the number of secondary servers.

8.1 Design

The basic idea of RTC is to execute the commit phase of a transaction in a dedicated main server core, and to detect non-conflicting pending transactions in another secondary server core. This way, if a processor contains n cores, two cores will be dedicated as servers, and the remaining $n-2$ cores will be assigned to clients. For this reason, RTC is more effective when the number of cores is large enough to afford dedicating two of them as servers. However, the core count in modern architectures is increasing, so that reserving two cores does not represent a limitation for RTC applicability.

RTC architecture can be considered as an extension of NOrec. Figure 8.1 shows the structure of a NOrec transaction. A transaction can be seen as the composition of three main parts: initialization, body, and commit. The initialization part adjusts the local variables at the beginning of the transaction. In the transaction body, a set of speculative reads and writes are executed. During each read, the local read-set is validated to detect conflicting writes of concurrent transactions, and, if the validation is successful, the new read is added to the read-set. Writes are also saved in local write-sets to be published at commit. During the commit phase, the read-set is repeatedly validated until the transaction acquires the lock (by an atomic CAS operation to increase the global timestamp). The write-set is then published into the shared memory, and finally, the global lock is released.

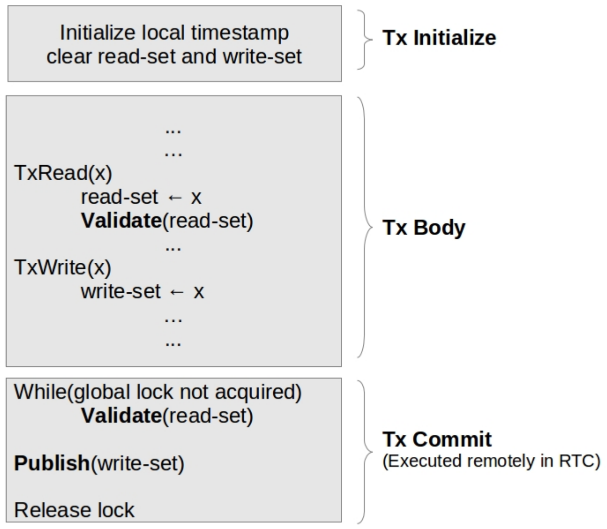


Figure 8.1: Structure of a NOrec transaction

This well defined mechanism of NOrec can be converted to remotely executing the transaction commit part. Unlike lock-based applications, which contain programmer-defined locks with generic critical sections, RTC knows precisely the number of locks to acquire (i.e., only one global lock), when to execute the commit (i.e., at the transaction end), and what to execute inside the commit (i.e., validating transaction read-set and publishing its write-set). This simplifies the role of the servers, in contrast to server-based optimizations for locks such as RCL [71] and Flat Combining [55], which need additional mechanisms (either by re-engineering as in RCL or at run-time as in Flat Combining) to indicate the procedures to execute in behalf of the clients to the servers.

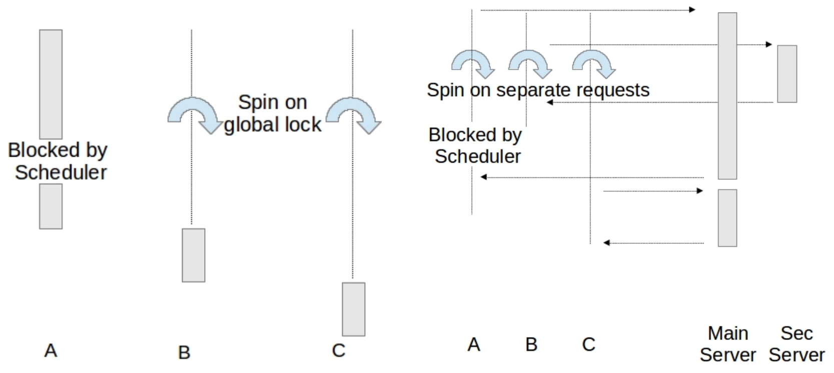
RTC is therefore simple: clients communicate with servers (either main or secondary) using a cache-aligned requests array to reduce caching overhead. A client's commit request always contains a reference to the transactional context (read-set and write-set, local timestamp, and bloom filters). This context is attached to the transaction request when it begins. A client starts its commit request by changing a *state* field in the request to a pending state, and then spins on this *state* field until the server finishes the execution of its commit and resets the *state* field again. On the server side, the main server loops on the array of commit requests until it finds a client with a pending state. The server then obtains the transaction's context and executes the commit. While the main server is executing a request, the secondary server also loops on the same array, searching for independent requests. Note that it does not execute any client requests unless the main server is executing another non-conflicting request.

Figure 8.2 illustrates the flow of execution in both NOrec and RTC. Assume we have three transactions. Transaction T_A is a long running transaction with a large write-set. Transaction T_B does not conflict with T_A and can be executed concurrently, while transaction T_C is conflicting with T_A. Figure 8.2(a) shows how NOrec executes these three transactions. If T_A acquires the lock first, then both T_B and T_C will spin on the shared lock until T_A completes its work and releases the lock, even if they can run concurrently. Spinning on the same lock results in significant number of useless CAS operations and cache misses. Moreover, if T_A is blocked by the OS scheduler, then both the spinning transactions will also wait until T_A resumes, paying an additional cost. This possibility of OS blocking increases with the number of busy-waiting transactions.

In Figure 8.2(b), RTC moves the execution of T_A's commit to the main server. Transactions T_B and T_C send a commit request to the server and then spin on their own requests (instead of spinning on a shared global lock) until they receive a reply. During T_A's execution, the secondary server (which is dedicated to detecting dependency) discovers that T_B can run concurrently with T_A, so it starts executing T_B without waiting for T_A to finish. Moreover, when T_A is blocked by the OS scheduler, this blocking does not affect the execution of its commit on the main server, and does not block other transactions. Blocking of the servers is much less frequent here, because client transactions are not allowed to execute on server cores.

8.1.1 Dependency Detection

RTC leverages a secondary server to solve the problem of unnecessary serialization of independent commit requests. The secondary server uses bloom filters [12] to detect dependency between transactions. Each transaction keeps two local bloom filters and updates them at each read/write (in addition to updating the normal read-set and write-set). The first one is a write filter, which represents the transaction writes, and the second one is a read-write filter, which represents the union of the transaction reads and writes. If the read-write filter



(a) NOrec (b) RTC

Figure 8.2: Flow of commit execution in NOrec and RTC.

of a transaction T_X does not intersect with the write filter of a transaction T_Y currently executed in the main server, then it is safe to execute T_X in the secondary server. (We provide a proof of the independence between T_X and T_Y in Section 8.3).

Synchronization between threads in NOrec is done using a single global sequence lock. Although RTC needs more complex synchronization between the main server and the secondary server, in addition to synchronization with the clients, we provide a lightweight synchronization mechanism that is basically based on the same global sequence lock, and one extra servers lock. This way, we retain the same simplicity of NOrec's synchronization. (Section 8.2 details RTC's synchronization).

The effectiveness of the secondary server is evident when the write-set size is large. The secondary server adds synchronization overhead to the main server. This overhead will be relatively small if commit phase is long, and transactions are mostly independent. On the other hand, if write-sets are short (indicating short commit phases), then even if transactions are independent, the time taken by the secondary server to detect such independent transactions is long enough so that the main server may finish its execution before the secondary server makes a substantial progress. To solve this issue, RTC dynamically enables/disables the secondary server according to the size of the transaction write-set. The secondary server works on detecting non-conflicting transactions when the write-set size exceeds a certain threshold (In Section 8.6, we show an experimental analysis of this threshold). As a consequence, the interactions between main and secondary servers are minimized so that the performance of the transactions executed in the main server (that represents the critical path in RTC) is not affected.

Another trade off for RTC is the bloom filter size. If it is too small, many false conflicts will occur and the detector will not be effective. On the other hand, large bloom filters need large time to be accessed. Bloom filter access must be fast enough to be fruitful. In our experiments, we used the same size as in the RSTM default configuration (1024 bits), as we found that other sizes give similar or worse performance.

8.1.2 Analysis of NOrec Commit Time

As we mentioned before, RTC is more effective if the commit phase is not too short. Table 8.1 provides an analysis of NOrec's commit time ratio of the STAMP benchmarks [81][3]. In this experiment, we measure the commit time as the sum of the time taken to acquire the lock and the time taken for executing the commit procedure itself. We calculated both *A)* the ratio of commit time to the transactions execution time (*%trans*), and *B)* the ratio of commit time to the total application time (*%total*). The results show that the commit time is already predominant in most of the STAMP benchmarks. The percentage of commit time increases when the number of threads increases (even if the *%total* decreases, which means that the non-transactional execution increases). This means that the overhead of acquiring the lock and executing commit becomes more significant in the transactional parts.

Benchmark	8 threads		16 threads		32 threads		48 threads	
	%trans	%total	%trans	%total	%trans	%total	%trans	%total
genome	49	32	53	14	54	5	56	3
intruder	25	19	37	31	39	26	19	9
kmeans	43	34	56	27	60	15	62	11
labyrinth	0	0	0	0	0	0	0	0
ssca2	83	53	94	63	95	66	92	39
vacation	6	5	17	16	42	36	50	45

Table 8.1: Ratio of NOrec's commit time in STAMP benchmarks.

As our experimental results show in Section 8.5, the increase of RTC performance is proportional to the commit latency. In fact, benchmarks with higher percentage of commit time (*genome*, *ssca2*, and *kmeans*) gain more from RTC than the others with small commit execution time (*intruder* and *vacation*) because the latter do not take advantages from the secondary server. However, they still gain from the main server, especially when the number of transactions increases and competition between transactions on the same lock becomes a significant overhead. Benchmarks with a dominating non-transactional workloads (*labyrinth*) show no difference between NOrec and RTC because no operations are done for those non-transactional parts during the commit phase.

[3]We excluded *yada* here and in all our further experiments as it evidenced errors (segmentation fault) when we tested them on RSTM, even in the already existing algorithms like NOrec. We also excluded *bayes* because its performance varies significantly between runs, so it is not useful for benchmarking.

8.2 RTC Algorithm

The main routines of RTC are servers loops and the new commit procedure. The initialization procedure and transaction body code are straightforward, thus we only briefly discuss them.

8.2.1 RTC Clients

Client commit requests are triggered using a cache-aligned requests array. Each commit request contains three items:

- **state.** This item has three values. READY means that the client is not currently executing commit. PENDING indicates a commit request that is not handled by a server yet. ABORTED is used by the server to inform the client that the transaction must abort.
- **Tx.** This is the reference to the client's transactional context. Basically, servers need the following information from the context of a transaction: *read-set* to be validated, *write-set* to be published, the *local timestamp*, which is used during validation, and *filters*, which are used by the secondary server.
- **pad.** This is used to align the request to the cache line (doing so decreases false sharing).

RTC initialization has two main obligations. The first one is allocating the requests array and starting the servers. The second one is to set the affinity of the servers to their reserved cores.

When a transaction begins, it is bound to the clients' *cpuset* to prevent execution on server cores (note that it's allowed to bound more than one client to the same core, according to the scheduler). It also inserts the reference of its context in the requests array. Finally, the local timestamp is assigned to the recent consistent global timestamp. Reads and writes update the bloom filters in addition to their trivial updates of read-sets and write-sets. Reads update the read-write filter, and writes update both the write filter and the read-write filter.

Client post validation is value-based like NOrec. Algorithm 21 shows how it generally works. In lines 3-4, transaction takes a snapshot of the global timestamp and loops until it becomes even (meaning that there is no commit phase currently running on both main and secondary servers). Then, read-set entries are validated (line 5). Finally, the global timestamp is read again to make sure that nothing is modified by another transaction while the transaction was validating (Lines 6-9).

Servers need also to validate the read-set before publishing the write-set. The main difference between server validation and client validation is that there is no need to check the timestamp by the server, because the main server is the only thread that changes the timestamp.

Algorithm 21 RTC: client validation

```
1: procedure CLIENT-VALIDATION
2:     $t = global\_timestamp$
3:     if $t$ is odd then
4:         retry validation
5:     Validate read-set values
6:     if $t \neq global\_timestamp$ then
7:         retry validation
8:     else
9:         return $t$
10: end procedure
```

Algorithm 22 RTC: client commit

```
1: procedure COMMIT
2:     if $read\_only$ then
3:         ...
4:     else
5:         if ¬ Client-Validate($Tx$) then
6:             TxAbort()
7:         $req.state$ = PENDING
8:         loop while $req.state \notin$ (READY, ABORTED)
9:         if $req.state$ = ABORTED then
10:            TxAbort()
11:        else
12:            TxCommit()
13:        ResetFilters()
14: end procedure
```

Algorithm 22 shows the commit procedure of RTC clients. Read-only transactions do not need to acquire any locks and their commit phase is straightforward. A read-write transaction starts its commit phase by validating its read-set to reduce the overhead on servers if it is already invalid (line 5). If validation succeeds, it changes its state to PENDING (line 7). Then it loops until one of the servers handles its commit request and changes the state to either READY or ABORTED (line 8). It will either commit or roll-back according to the reply (lines 9–12). Finally, the transaction clears its bloom filters for reusing them (line 13).

8.2.2 Main Server

The main server is responsible for executing the commit part of any pending transaction. Algorithm 23 shows the main server loop. By default, the dependency detection (DD) is disabled. The main server keeps looping on client requests until it reaches a PENDING request (line 6). Then it validates the client read-set. If validation fails, the server changes the state to ABORTED and continues searching for another request. If validation succeeds, it starts the commit operation in either DD-enabled or DD-disabled mode according to a threshold of the client write-set size (lines 7–14).

Execution of the commit phase without enabling DD is straightforward. The timestamp is increased (which becomes odd, indicating that the servers are working), the write-set is published to memory, the timestamp is then increased again (to be even), and finally the request state is modified to be READY.

Algorithm 23 Main server loop. Server commit with dependency detection disabled/enabled.

```
 1: procedure MAIN SERVER LOOP
 2:     $DD = false$
 3:     while $true$ do
 4:         for $i \leftarrow 1, num\_transactions$ do
 5:             $req \leftarrow req\_array[i]$
 6:             if $req.state$ = PENDING then
 7:                 if $\neg$ Server-Validate($req.Tx$) then
 8:                     $req.state$ = ABORTED
 9:                 else if $write\_set\_size < \mathrm{t}$ then
10:                     Commit(DD-Disabled)
11:                 else
12:                     $DD = true$
13:                     Commit(DD-Enabled)
14:                     $DD = false$
15: end procedure
16: procedure COMMIT(DD-Disabled, req)
17:     $global\_timestamp{+}{+}$
18:     WriteInMemory($req.Tx.writes$)
19:     $global\_timestamp{+}{+}$
20:     $req.state$ = READY
21: end procedure

22: procedure COMMIT(DD-Enabled, req, i)
23:     $mainreq = req\_array[i]$
24:     $global\_timestamp{+}{+}$
25:     WriteInMemory($req.Tx.writes$)
26:     loop while !CAS($servers\_lock$, $false$, $true$)
27:     $global\_timestamp{+}{+}$
28:     $mainreq$ = NULL
29:     $servers\_lock = false$
30:     $req.state$ = READY
31: end procedure
```

When DD is enabled, synchronization between the servers is handled using a shared *servers_lock*. First, the main server informs the secondary server about its current request number (line 23). Then, the global timestamp is increased (line 24). The order of these two lines is important to ensure synchronization between the main and secondary servers. The main server must also acquire *servers_lock* before it increments the timestamp again at the end of the commit phase (lines 26–29) to prevent the main server from proceeding until the secondary server finishes its work. As we will show in the correctness part (in Section 8.3), this *servers_lock* guarantees that the secondary server will only execute as an extension of the main server's execution. Comparing the two algorithms, we see that DD adds only one CAS operation on a variable, which is (only) shared between the servers. Also, DD is not enabled unless the write-set size exceeds the threshold. Thus, the overhead of DD is minimal.

8.2.3 Secondary Server

Algorithm 24 shows the secondary server's operation. It behaves similar to the main server except that it does not handle PENDING requests unless it detects that:

- DD is enabled (line 4);
- Timestamp is odd, which means that main server is executing a commit request (line 6);
- The new request is independent from the current request handled by the main server (line 9).

Algorithm 24 RTC: secondary server

```
1:  procedure SECONDARY SERVER LOOP
2:      while $true$ do
3:          for $i \leftarrow 1, num\_transactions$ do
4:              if $DD$ = false then continue
5:              $s = global\_timestamp$
6:              if $s\&1 = 0$ then continue
7:              $req \leftarrow req\_array[i]$
8:              if $req.state$ = PENDING then
9:                  if Independent($req$,$mainreq$) then
10:                     Commit(Secondary)
11: end procedure

12: procedure COMMIT SECONDARY(req)
13:     if $\neg$ Server-Validate($req.Tx$) then
14:         $aborted = true$
15:     if CAS($servers\_lock$, $false$, $true$) then
16:         if $s <> global\_timestamp$ or $mainreq$ = NULL then
17:             $servers\_lock = false$
18:         else if $aborted = true$ then
19:             $req.state$ = ABORTED
20:             $servers\_lock = false$
21:         else
22:             WriteInMemory($req.Tx.writes$)
23:             $req.state$ = READY
24:             $servers\_lock = false$
25:             loop while $global\_timestamp = s$
26: end procedure
```

The commit procedure is shown in lines 12–26. Validation is done before acquiring $servers\_lock$ to reduce the time of holding the lock (lines 13–14). However, since it is running concurrently with the main server, the secondary server has to validate that the main server is still handling the same request (line 16) after acquiring $servers\_lock$. This means that, even if the secondary server reads any false information from the above three points, it will detect that by observing either a different timestamp or a NULL $mainreq$ after the acquisition of $servers\_lock$. The next step is to either commit or abort according to its earlier validation (lines 18-24). Finally, in case of commit, secondary server loops until the main server finishes its execution and increases the timestamp (line 25). This is important to prevent handling another request, which may be independent from the main server's request but not independent from the earlier request.

The secondary server does not need to change the global timestamp. Only the main server increases it at the beginning and at the end of its execution. All pending clients will not make any progress until the main server changes the timestamp to an even number, and the main server will not do so until the secondary server finishes its work (because if the secondary server is executing a commit phase, it will be holding $servers\_lock$).

8.3 Correctness

To prove the correctness of RTC, we first show that there are no race conditions impacting RTC's correctness when the secondary server is disabled. Then, we prove that our approach of using bloom filters guarantees that transactions executed on the main and secondary servers are independent. Finally, we show how adding a secondary server does not affect race-freedom between the main and the secondary server, or between clients and servers [4].

RTC with DD Disabled: With the secondary server disabled, RTC correctness is similar to that of NOrec. Briefly, the post validation in Algorithm 21 ensures that: *i)* no client is validating while server is committing, and *ii)* each transaction sees a consistent state after each read. The only difference between NOrec and RTC without dependency detection is in the way they increment the timestamp. Unlike NOrec, there is no need to use the CAS operation to increase the global timestamp, because no thread is increasing it except the main server. All commit phases are executed serially on the main server, which guarantees no write conflicts during commit, either on the timestamp or on the memory locations themselves.

Transaction Independence: The secondary server adds the possibility of executing two independent transactions concurrently. To achieve that, each transaction keeps two bloom filters locally: a write-filter "$wf(t)$", which is a bitwise representation of the transaction write-set, and a read-write filter "$rwf(t)$", which represents the union of its read-set and write-set. Concurrent commit routines (in both main and secondary servers) are guaranteed to be independent using these bloom filters. We can state that: if a transaction T_1 is running on the RTC main server, and there is a pending transaction T_2 such that $rwf(T_2) \cap wf(T_1) = \emptyset$, then T_2 is independent from T_1 and can run concurrently using the secondary server. This can be proven as follows: T_1 does not increase the timestamp unless it finishes validation of its read-set. Thus, T_2 will not start unless T_1 is guaranteed to commit. Since $rwf(T_2) \cap wf(T_1) = \emptyset$, T_1 can be serialized before T_2. T_1 cannot invalidate T_2 because T_1's write-set does not intersect with T_2's read-set. The write-after-write hazard also cannot

[4]In all of the proof arguments, we assume that instructions are executed in the same order as shown in Section 8.2's algorithms – i.e., sequential consistency is assumed. We ensure this in our C/C++ implementation by using memory fence instructions when necessary (to prevent out-of-order execution), and by using volatile variables when necessary (to prevent compiler re-ordering).

happen because the write filters are not intersecting. If T_2 aborts because of an invalidated read-set, it will not affect T_1's execution.

RTC with DD Enabled: Finally, we prove that transaction execution is still race-free when the secondary server is enabled. Synchronization between the main and the secondary server is guaranteed using the *servers_lock*. The main server acquires the lock before finishing the transaction (clearing *mainreq* and incrementing the global timestamp) to ensure that the secondary server is idle. The secondary server acquires the *servers_lock* before starting, and then it validates both *mainreq* and the timestamp. If they are invalid, the secondary server will not continue, and will release the *servers_lock*, because it means that the main server finishes its work and starts to search for another request.

More detailed, in lines 26-29 of Algorithm 23, the main server increments the timestamp in a mutually exclusive way with lines 15-24 of the secondary server execution in Algorithm 24 (because both parts are enclosed by acquiring and releasing the *servers_lock*). Following all possible race conditions between line 27 of Algorithm 23 and the execution of the secondary server shows that the servers' executions are race-free. Specifically, there are four possible cases for the secondary server when the main server reaches line 27 (incrementing the timestamp after finishing execution):

- Case 1: before the secondary server takes a snapshot of the global timestamp (before line 5). In this case, the secondary server will detect that the main server is no longer executing any commit phase. This is because, the secondary server will read the new (even) timestamp, and will not continue because of the validation in line 6.

- Case 2: after the secondary server takes the snapshot and before it acquires the *servers_lock* (after line 5 and before line 15). In this case, whatever the secondary server will detect during this period, once it tries to acquire the *servers_lock*, it will wait for the main server to release it. This means that, after the secondary server acquires the *servers_lock*, it will detect that the timestamp is changed (line 16) and it will not continue.

- Case 3: after the secondary server acquires the *servers_lock* and before this lock is released (after line 15 and before line 24). This cannot happen because the *servers_lock* guarantees that these two parts are mutually exclusive. So, in this case, the main server will keep looping until the secondary server finishes execution and releases the *servers_lock*.

- Case 4: after the secondary server releases the *servers_lock* (after line 24). This scenario is the only scenario in which the main and secondary servers are executing commit phases concurrently. Figure 8.3 shows this scenario. In this case, the secondary server works only as an extension of the currently executed transaction in the main server, and the main server cannot finish execution and increment the timestamp unless the secondary server also finishes execution.

Thus, the main server will not continue searching for another request until the secondary server finishes its execution of any independent request. Line 25 of Algorithm 24 guarantees the same behavior in the other direction. If the secondary server finishes execution first, it will keep looping until the main server also finishes its execution and increments the timestamp (which means that the secondary server executes only one independent commit request per each commit execution on the main server).

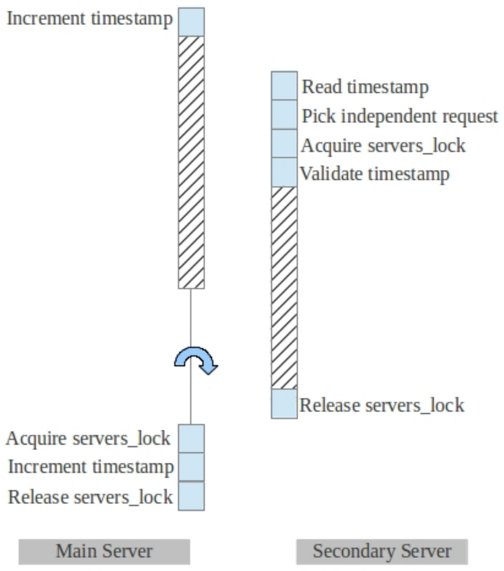


Figure 8.3: Flow of commit execution in the main and secondary servers. Even if the main server finishes execution before the secondary server, it will wait until the secondary server releases the $servers\_lock$.

In conclusion, the RTC servers provide the same semantics of single lock STM algorithms. Although two independent commit blocks can be concurrently executed on the main and secondary servers, they appear to other transactions as if they are only one transaction because they are synchronized using different locks (not the global timestamp). Figure 8.3 shows that the global timestamp increment (to be odd and then even) encapsulates the execution of both the main and the secondary servers. This also guarantees that the clients' validations will not have race-conditions with the secondary server, because clients are not validating in the period when the timestamp is odd.

Using a single lock means that RTC is privatization-safe, because writes are atomically published at commit time. The value-based validation minimizes false conflicts and enables the detection of non-transactional writes. Finally, servers repeatedly iterate on clients requests, which guarantees livelock-freedom and more fair progress of transactions.

8.4 RTC with Flat-Combining

The approach of dedicating cores for executing server threads has its own cost because it disallows cores from running application threads and it enforces data accessed by transactions to be cached on those cores. As we show later in Section 8.5, in most workloads this cost is dominated by the benefits of reducing cache misses, reducing CAS operations, and preventing lock holders from being descheduled. However, in some architectures, especially those with low core count, this cost may become notable and impact the performance. To address this issue, we extend RTC by introducing RTC-FC, a version of RTC with no dedicated cores for servers.

The only distinguishing point between RTC-FC and RTC is on the assignment of the thread that plays the role of the server. In RTC-FC, we use an idea similar to *flat combining* [55] that selects one of the running clients to combine the requests of pending clients. To do that, each client changes its request status from `READY` to `PENDING`, and then it tries to be the combiner (using one CAS operation on a `combiners-lock`). If it succeeds, it executes one server iteration, serving all the `PENDING` requests, including its own request, similar to an iteration of Algorithm 23 with *DD-DISABLED*. If the CAS fails, this means that another thread became the combiner, thus, the thread spins on its request status similarly to RTC. During its spinning, it periodically checks if the current combiner releases the `combiners-lock`. If so, it retries to be the combiner.

Although RTC-FC avoids dedicating cores for servers, it has the following overheads: first, it adds at least one more CAS operation for each transaction; second, it increases the probability of descheduling the `combiners-lock` holder; and finally, it obligates threads to pause their executions while servicing other requests, which also adds an overhead due to caching the data of other requests. The problem of descheduling a combiner can be partially solved by enforcing the clients that fail to CAS the `combiners-lock` to call *sched_yield* in order to give up their OS time slice instead of spinning. The effect of overloading the combiner cache with the data of other transactions can also be alleviated by using a NUMA-aware flat-combining algorithm similar to [36], where the combiner executes the commit phases of transactions belonging only to its NUMA-zone in order to exploit the locality of this NUMA-zone. In the next section we show how those parameters affect the performance of RTC-FC, and discuss the cases in which RTC-FC fits best.

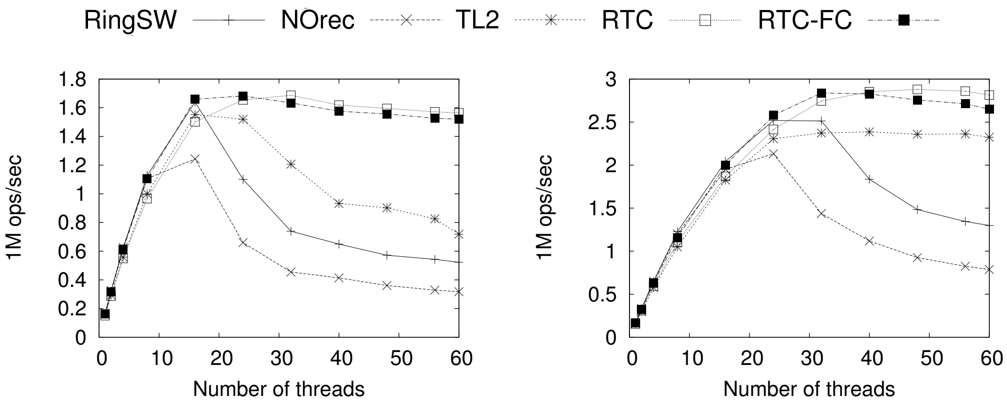
8.5 Experimental Evaluation

We implemented RTC in C++ (compiled with gcc 4.6) and integrated into the RSTM framework [73] (compiled using default configurations). Our experiments are performed on a 64-core AMD Opteron machine (128GB RAM, 2.2 GHz, 64K of L1 cache, 2M of L2 cache) with 4 sockets and 16 cores per socket (with 2 NUMA-zones per physical socket, making a total of 8 NUMA-zones). Each NUMA-zone has a 6M of L3 cache.

Our benchmarks for evaluation included micro-benchmarks (such as red-black tree and linked list), and the STAMP benchmark suite [81]. We also evaluated multi-programming cases, where the number of transactions is more than the number of cores. Our competitors include NOrec (as representative of approaches relying on global metadata), RingSW (because it uses bloom filter), and TL2 (representing an ownership-record based approach). We used a privatization-safe version of TL2 for a fair comparison. All the STM algorithms and the benchmarks used are those available in RSTM. All reported data points are averages of 5 runs.

8.5.1 Red-Black Tree

Figure 8.4 shows the throughput of RTC, RTC-FC, and their competitors on a red-black tree with 1M elements and a delay of 100 no-ops between transactions. In Figure 8.4(a), when 50% of operations are reads, all algorithms scale similarly (RTC-FC is slightly better than the others), but both versions of RTC sustain high throughput, while other algorithms' throughput degrades. This is a direct result of the cache-aligned communication among transactions. In high thread count, RTC is slightly better than RTC-FC because of the overheads of the latter as discussed in Section 8.4. In Figure 8.4(b), when 80% of the operations are reads, the degradation point of all algorithms shifts (to the right) because contention is lower. However, RTC scales better and reaches peak performance when contention increases. At high thread count, RTC improves over the best competitor by 60% in the first case and 20% in the second one.



(a) 50% reads (b) 80% reads

Figure 8.4: Throughput (per micro-second) on red-black tree with 1M elements.

Additionally, we focused on making a connection between the performance (in terms of throughput) and the average number of cache misses per transaction generate by NOrec and RTC. Figure 8.5 shows the results. At high number of threads, the number of cache

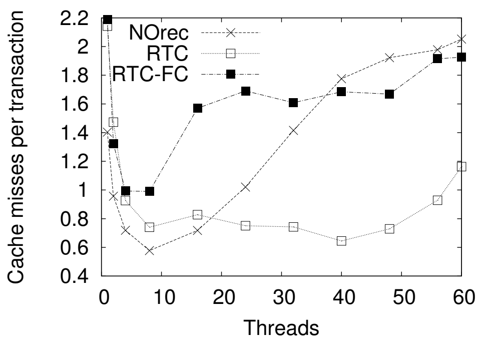


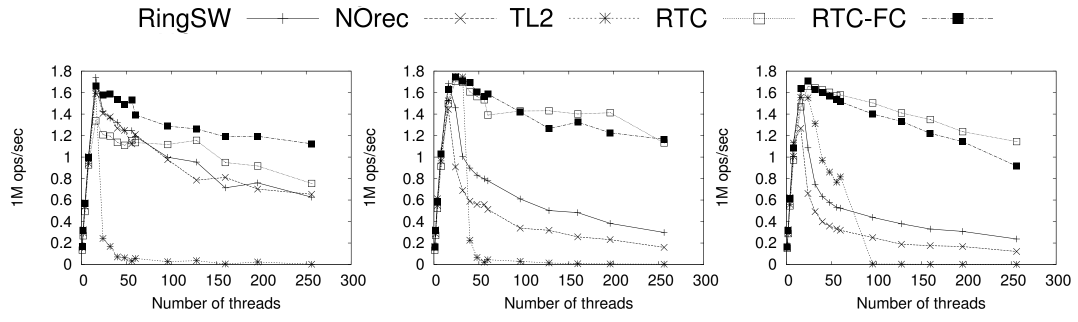
Figure 8.5: Cache misses per transaction on red-black tree with 1M elements.

misses per transaction on NOrec is higher than RTC. Comparing Figures 8.4(a) and 8.5, an interesting connection can be found between the point in which the performance of NOrec starts to drop and the point in which the number of NOrec's cache misses starts to increase. This comparison clearly points out the impact of RTC design on decreasing cache misses due to spin locks. RTC-FC on the other hand, suffers from more cache-misses. However, those misses are not generated because of spinning. Rather, they are generated because each client is playing the role of the server frequently, and thus it is obligated to validate the read-set of the other clients, which may result in evicting its own data from its cache. Summarizing, although the number of cache misses is not the only parameter that affects the performance of RTC, Figure 8.5 gives an important reasoning about the effect of RTC and RTC-FC on the cache misses on both the actual data and the meta-data. Such a comparison allows for a better understanding of their behavior.

In the next experiment, we created up to 256 threads[5] and repeated the experiment while progressively enabling the cores of only one socket (16 cores), two sockets (32 cores), and the whole four sockets (64 cores). Our goal in this experiment is to make a stress test to reveal the side effect of having server-client communication in heavy loads, as well as to seek RTC's performance saturation point. More specifically, we use the approach of enabling/disabling CPU sockets to analyze the effect of *i)* running very large number of concurrent threads on few number of cores, while dedicating two of them as servers, and *ii)* having inter-socket synchronization rather than intra-socket synchronization.

Figure 8.6 shows the results in a red-black tree with 50% reads. In Figure 8.6(a), when only one socket is enabled, all the transactions execute on one socket, which decreases the overhead of cache misses and CAS operations. For this reason, both versions of RTC cannot gain a lot from the efficient remote core locking mechanism, and thus the gap between them and the other algorithms is small. Additionally, dedicating two cores out of sixteen as servers in RTC has a significant effect on the overall performance. That is why in this case, RTC-FC performs better than RTC. In Figure 8.6(b), when the number of cores becomes 32 (on 2

[5]This is the maximum number of threads allowed by RSTM.



(a) 16 cores - 1 socket (b) 32 cores - 2 sockets (c) 64 cores - 4 sockets

Figure 8.6: Throughput on red-black tree with 1M elements, 100 no-ops between transactions, 50% reads.

sockets), the penalty of dedicating two cores for RTC decreases, thus RTC and RTC-FC perform similarly. At the same time, the overheads in the other algorithms increase because meta-data now are cached in two sockets rather than one. As a result, the overall performance of RTC/RTC-FC increases compared to the other STM algorithms. The performance improvement continues in the last case (Figure 8.6(c)), when the number of cores becomes 64 (on 4 sockets). Specifically, starting from 32 threads, RTC/RTC-FC perform better than the best competitor by an average of 3x at high thread count. Our analysis confirms previous studies, which conclude that cross-socket sharing should be minimized as it is one of the performance killers [30]. Also, in this case, RTC becomes better than RTC-FC because the overhead of dedicating cores is minimized while the overhead of overloading the clients with the combiner tasks increases.

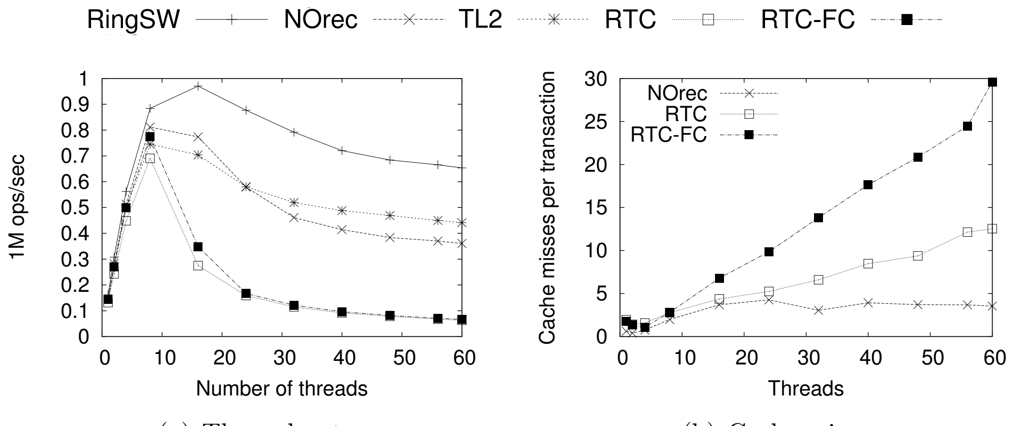
Figure 8.6 also shows that in the multi-programming case (when threads become more than cores), RTC's performance starts to slightly degrade like the other STM algorithms. However, this degradation is the normal degradation due to the contention on the shared red-black tree, which confirms that RTC solves the issue of spin locking and leaves the overhead of STM framework limited to the contention on the application-level data.

To conclude, RTC still has some limitations, like the effect of dedicating cores for servers, and the normal contention on the application-level data (which is not targeted by RTC's mechanism). However, when the number of cores increases, these negative effects are dominated by the improvements due to the optimized locking mechanism.

8.5.2 Linked List

In Figure 8.7(a), we show the results using the linked list benchmark. It represents the worst case workload for RTC/RTC-FC and we include it in order to show the RTC design's limitations in unfavorable scenarios. Linked list is a benchmark which exposes non optimal characteristics in terms of validation and commit. In fact, in a doubly linked list with only 500 nodes, each transaction makes on average hundreds of reads to traverse the list, and then it executes few writes (two writes in our case) to add or remove the new node. This means that the read-set size is too large compared to the write-set size. Since RTC servers have to re-validate the read-set of the clients before publishing the write-set, pulling a large read-set like that affects the performance significantly and nullifies any gains from optimizing the actual commit phase (which mainly consists of acquiring the global timestamp and publishing the write-set). Figure 8.7(b) confirms that by showing the cache-misses per transaction in that case. Here, the cache misses saved by the cache-aligned communication are clearly dominated by thrashing the cache of the servers with the read-sets of the clients. The problem increases in RTC-FC since the combiners are actual client threads that may sacrifice their own cached data to validate the reads-set of the other clients. It is worth to note that this issue does not occur for read-only workloads, because RTC does not involve servers in executing read-only transactions.

As a solution to this issue, an STM runtime can be made to heuristically detect these cases of RTC degradation by comparing the sizes of read-sets and write-sets, and switching at run-time from/to another appropriate algorithm as needed. Earlier work proposes a lightweight adaptive STM framework [96]. In this framework, the switch between algorithms is done in a "stop-the world" manner, in which new transactions are blocked from starting until the current in-flight transactions commit (or abort) and then switch takes place. RTC can be easily integrated in such a framework. Switching to RTC only requires allocating the requests array and binding the servers and clients to their *cpusets* (which can be achieved using C/C++ APIs). Switching away from RTC requires terminating the server threads and deallocating the requests array.



(a) Throughput (b) Cache-misses

Figure 8.7: Throughput and cache-misses per transaction on doubly linked list with 500 elements, 50% reads, and 100 no-ops between transactions.

8.5.3 STAMP

Figure 8.8 shows the results for six STAMP benchmarks, which represent more realistic workloads with different attributes. It is important to relate these results to the commit time analysis in Table 8.1. RTC has more impact when commit time is relatively large, especially when the number of threads increases.

In four out of these six benchmarks (*ssca2*, *kmeans*, *genome*, and *labyrinth*), RTC has the best performance when the number of threads exceeds 20. Moreover, for *kmeans* and *ssca2*, which have the largest commit overhead according to Table 8.1, RTC has better performance than all algorithms even at low number of threads. For *labyrinth*, RTC performs the same as NOrec and TL2, and better than RingSW. This is because, RTC does not have any overhead on non-transactional parts, which dominate in *labyrinth.* Also, in all cases, RTC outperforms NOrec at high number of threads. Even for *vacation* and *intruder*, where the commit time percentage is small (6%–50% and 19%–39%, respectively), RTC outperforms NOrec and has the same performance as RingSW for high number of cores. In those two benchmarks, TL2 is the best algorithm (especially for low thread count) because they represent low-contention workloads, where serializing the (mostly independent) commit phases, as in NOrec and RTC, affects the performance. For RTC specifically, like linked list, scalability in those benchmarks is clearly affected by the small ratio of the commit phase (as mentioned in Table 8.1).

In general, both RTC and RTC-FC perform similarly. However, RTC-FC performs better in the low-contention workloads (e.g., *genome* and *vacation*), especially for small thread count (less than 8). This gap decreases at high number of threads (more than 8), because the benefit of dedicating cores for servers in RTC (e.g., disallowing lock holder descheduling) increases.

8.6 Extending RTC with more servers

The current implementation of RTC uses only two servers: one main server and one secondary server. It is easy to show that using one main server is reasonable. This is because we replace only one global lock (in NOrec) with a remote execution. Even if we add more main servers, their executions will be serialized because of this global locking. Adding more secondary servers (which search for independent requests) is, however, reasonable. This is because it may increase the probability of finding such an independent request in a reasonable time, which increases the benefits from secondary servers. However, leveraging on a fine grain performance analysis, we decided to tune RTC with only one secondary server. This decision is supported by the results obtained by running RTC with more secondary servers, which are shown in Figure 8.9. They highlight that the synchronization overhead needed for managing the concurrent execution of more secondary servers, is higher than the gain achieved.

In Figure 8.9, N reads and N writes of a large array's elements are executed in a transaction,

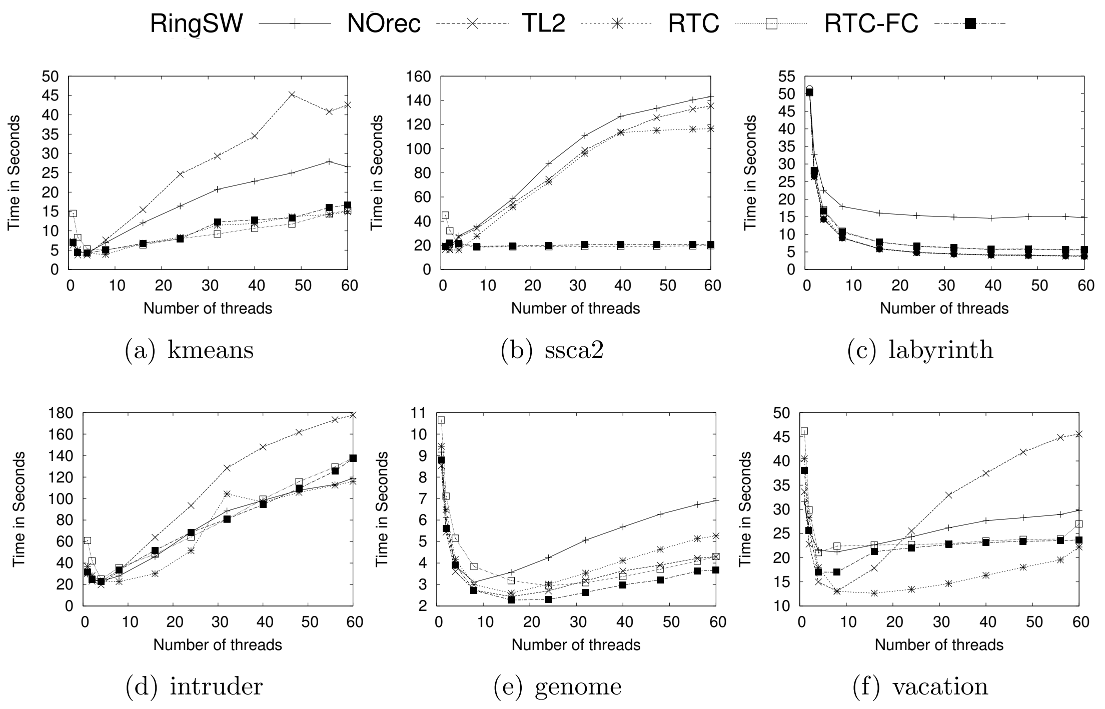


Figure 8.8: Execution time on STAMP benchmark suite's applications.

which results in write-sets of size N. The writes are either totally dependent by enforcing at least one write to be shared among transactions, or independent by making totally random reads and writes in a very long array. Figure 8.9 shows that the overhead of adding another secondary server is more than its gain. Performance enhancement using one DD is either the same or even better than using two DD in all cases. The same conclusion holds for executing more than one commit phase on the secondary server in parallel with the same main server's commit. Although both enhancements should have a positive effect in some theoretical cases of very long main server commits, we believe that in practical cases, like what we analyzed, the gain is limited.

We also used this experiment to determine the best threshold of the write-set size after which we should enable dependency detection. The time taken by the main server to finish the commit phase is proportional to the size of the write-set (read-set size is not a parameter because validation is made before increasing the timestamp). Thus, small write-sets will not allow the secondary server to work efficiently and will likely add unnecessary overhead (putting into consideration that the time taken by the secondary server to detect independent transactions does not depend on the transaction size because bloom filters are of constant size and they are scanned in almost constant time). To solve this problem, RTC activates the secondary server only when the size of the write-set exceeds a certain threshold.

In case of dependent transactions, the dependency detector (DD) cannot enhance performance because it will not detect a single independent transaction. Note that, the overhead of DD does not exceed 5% though, and it also decreases when the write-set size increases (reaches 0.5% when the size is 50). When transactions are independent, DD starts to yield significant performance improvement when the write-set size reaches 20 elements (obtains 30% improvement when size is 40). Before 10 elements, DD's overhead is larger than the gain from concurrent execution, which results in an overall performance degradation.

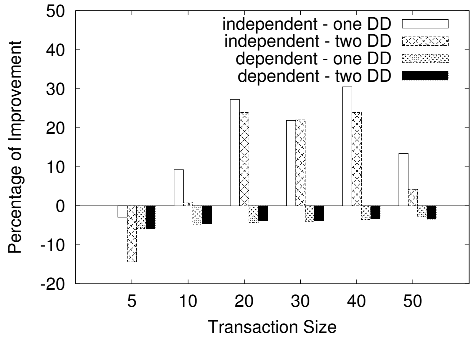


Figure 8.9: Effect of adding dependency detector servers.

We also calculated the number of transactions which are executed on the secondary server. In all the independent cases, it varies from 2% to 11%. Since the percentage of improvement is higher in most cases, this means that DD also saves extra time by selecting the most appropriate transaction to execute among the pending transactions, which reduces the probability of abort.

According to these results, we use only one secondary server, and we select a threshold of 20 elements to enable the secondary server in our experiments.

8.7 Using RTC in Hybrid TM

NOrec has been successfully used as a fallback path to *best-effort* HTM transactions [26, 91, 74]. This is because it uses only one global timestamp as a shared meta-data. Replacing NOrec with RTC in such hybrid algorithms is a feasible extension to our work. To do so, no modification is needed at the client (software) execution because HTM transactions only need to know whether there is a software transaction executing its commit phase or not (which would be done by the servers exploiting the global timestamp). Moreover, centralizing the commit phases in the servers allows for more optimizations on the hybrid algorithms themselves, such as exploiting servers for profiling the HTM execution.

8.8 Summary

Software transactional memory is a highly promising synchronization abstraction, but state-of-the-art STM algorithms are plagued by performance and scalability challenges. Analysis of these STM algorithms on the STAMP benchmark suite shows that transaction commit phases are one of the main sources of STM overhead. In this chapter we propose Remote Transaction Commit. RTC reduces this overhead with a simple idea: execute the commit phase in a dedicated servicing thread. This reduces cache misses, spinning on locks, CAS operations, and thread blocking. Our implementation and evaluation shows that the idea is very effective – up to 4x improvement over state-of-the-art STM algorithms in high thread count.

RTC builds upon similar ideas on remote/server thread execution previously studied in the literature, most notably, Flat Combining and RCL. However, one cannot simply apply them to an STM framework as is. In one sense, our work shows that, this line of reasoning is effective for improving STM performance.

Chapter 9

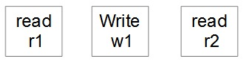
Remote Invalidation

In this chapter, we present *Remote Invalidation* (RInval), an STM algorithm which applies the same principles of RTC on invalidation-based STM algorithms (e.g., InvalSTM [40]). RTC and RInval share the advantage of reducing the locking overhead during the execution of STM transactions. However, locking overhead is not the only overhead in the critical path of the transaction (i.e., the sequence of operations that compose the execution of the transaction). Validation and commit routines themselves are overheads that sometimes interfere with each other. As we discussed in Section 2.1.2, invalidation-based algorithms, such as InvalSTM, are useful when the overhead of validation is significant because the time complexity of validation is reduced from a quadratic function to a linear function (in terms of the size of the read-set). However, InvalSTM adds a significant overhead on the commit routine, which affects the performance on many other cases. In this chapter, we present a comprehensive study on the parameters that affect the critical path of the transaction, and we study specifically the tradeoff between the validation and commit overheads. Then, we show how RInval significantly optimizes the transaction critical path.

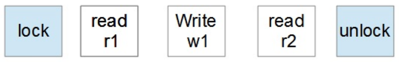
9.1 Transaction Critical Path

As described in Section 2.1.2, InvalSTM serializes commit and invalidation in the same commit routine, which significantly affects invalidation in a number of cases and degrades performance (we show this later in this section). Motivated by this observation, we study the overheads that affect the critical path of transaction execution to understand how to balance the overheads and reduce their effect on the critical path.

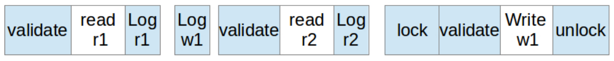
First, we define the *critical path* of a transaction as the sequence of steps that the transaction takes (including both shared memory and meta-data accesses) to complete its execution. Figure 9.1 shows this path in STM, and compares it with that in sequential execution and coarse-grained locking.



(a) Sequential code



(b) Coarse-grained locking



(c) STM

Figure 9.1: Critical path of execution for: (a) sequential, (b) lock-based, and (c) STM-based code

In Figure 9.1(a), the sequential code contains only shared-memory reads and writes without any overhead. Coarse-grained locking, in Figure 9.1(b), adds only the overhead of acquiring and releasing a global lock at the beginning and at the end of execution, respectively. However, coarse-grained locking does not scale and ends up with a performance similar to sequential code. It is important to note that fine-grained locking and lock-free synchronization have been proposed in the literature to overcome coarse-grained locking's scalability limitation [59]. However, these synchronization techniques must be custom-designed for a given application situation. In contrast, STM is a general purpose framework that is completely transparent to application logic. In application-specific approaches, the critical path cannot be easily identified because it depends on the logic of the application at hand.

Figure 9.1(c) shows how STM algorithms[1] add significant overheads on the critical path in order to combine the two benefits of *i)* being as generic as possible and *ii)* exploiting as much concurrency as possible. We can classify these overheads as follows:

Logging. Each read and write operation must be logged in local (memory) structures, usually called read-sets and write-sets. This overhead cannot be avoided, but can be minimized by efficient implementation of the structures [59].

Locking. We already studied this overhead when we discussed RTC in Chapter 8, which can be concluded in *i)* time of locking, *ii)* granularity of locking, and *iii)* locking mechanism. RInval follows the same RTC's guidelines to alleviate this overhead.

Validation. As mentioned before, the validation overhead becomes significant when higher levels of correctness guarantees are required (e.g., opacity). Most STM algorithms use either incremental validation or commit-time invalidation to guarantee opacity. In the case of invalidation, the time complexity is reduced, but with an additional overhead on commit, as

[1]Here, we sketch the critical path of NOrec [25]. However, the same idea can be applied to most STM algorithms.

we discuss in the next point.

Commit. Commit routines handle a number of issues in addition to publishing write-sets on shared memory. One of these issues is lock acquisition, which, in most STM algorithms, is delayed until commit. Also, most STM algorithms require commit-time validation after lock acquisition to ensure that nothing happened when the locks were acquired. In case of commit-time invalidation, the entire invalidation overhead is added to the commit routines. This means that a committing transaction has to traverse all active transactions to detect which of them is conflicting. As a consequence, the lock holding time is increased. Moreover, if the committing transaction is blocked for any reason (e.g., due to OS scheduling), all other transactions must wait. The probability of such blocking increases if the time of holding the lock increases. Therefore, optimizing the commit routines has a significant impact on overall performance.

Abort. If there is a conflict between two transactions, one of them has to abort. Transaction abort is a significant overhead on the transaction's critical path. The contention manager is usually responsible for decreasing the abort overhead by selecting the best candidate transaction to abort. The greater the information that is given to the contention manager from the transactions, the greater the effectiveness on reducing the abort overhead. However, involving the contention manager to make complex decisions adds more overhead to the transaction critical path.

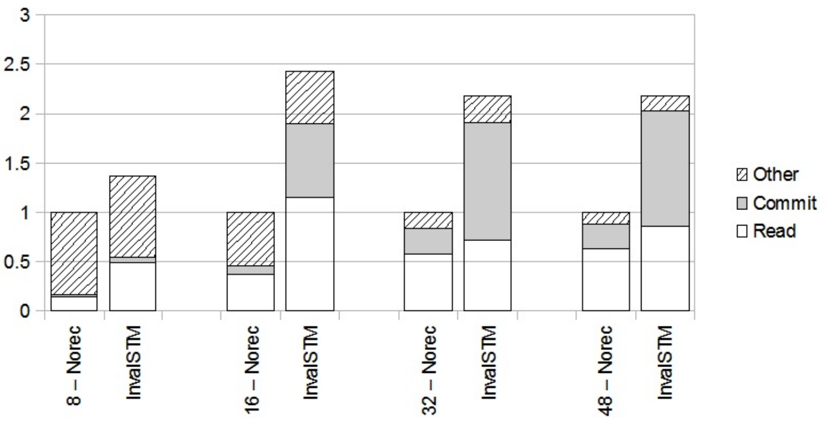


Figure 9.2: Percentage of validation, commit, and other (non-transactional) overheads on a red-black tree. The y-axis is the normalized (to NOrec) execution time

Figure 9.2 shows how the trade-off between invalidation and commit affects the performance in a red-black tree benchmark for different numbers of threads (8, 16, 32, and 48). Here, transactions are represented by three main blocks: read (including validation), commit (including lock acquisition, and also invalidation in the case of InvalSTM), and other overhead. The last overhead is mainly the non-transactional processing. Although some transactional work is also included in the later block, such as beginning a new transaction and logging writes, all of these overheads are negligible compared to validation, commit, and non-transactional overheads. Figure 9.2 shows the percentage of these blocks in both NOrec and InvalSTM (normalized to NOrec).

The figure provides several key insights. When the number of threads increases, the percentage of non-transactional work decreases, which means that the overhead of contention starts to dominate and becomes the most important to mitigate. It is clear also from the figure that InvalSTM adds more overhead on commit so that the percentage of execution time consumed by the commit routine is higher than NOrec. Moreover, this degradation in commit performance affects read operations as well, because readers have to wait for any running commit to finish execution.

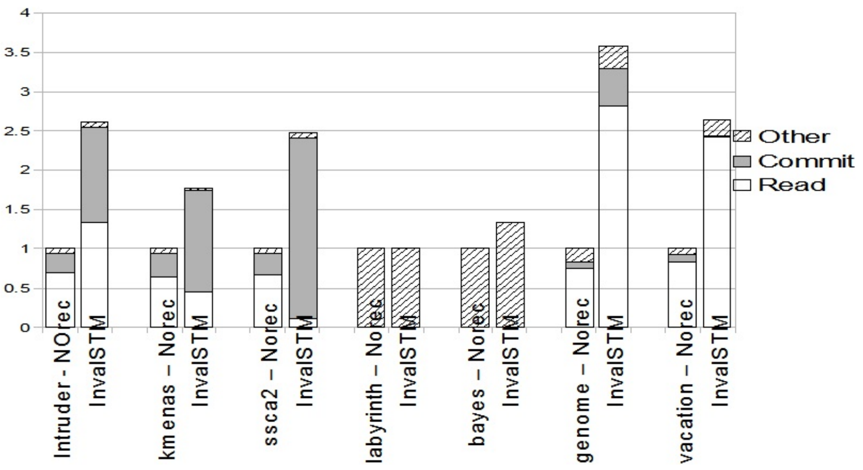


Figure 9.3: Percentage of validation, commit, and other (non-transactional) overheads on STAMP benchmark. The y-axis is the normalized (to NOrec) execution time

The same conclusion is given in the STAMP benchmark. In Figure 9.3, the percentage of commit in *intruder*, *kmeans*, and *ssca2*, is higher in InvalSTM than NOrec, leading to the same performance degradation as red-black tree. In *genome* and *vacation*, degradation in InvalSTM read performance is much higher than before. This is because these workloads are biased to generate more read operations than writes. When a committing transaction invalidates many read transactions, all of these aborted transactions will retry executing all of their reads again. Thus, in these read-intensive benchmarks, abort is a dominating overhead. In *labyrinth* and *bayes*, almost all of the work is non-transactional, which implies that using any STM algorithm will result in almost the same performance.

Based on this analysis, it is clear that each overhead cannot be completely avoided. Different STM algorithms differ on how they control these overheads. It is also clear that some overheads contradict each other, such as validation and commit overheads. The goal in such cases should be finding the best trade-off between them. This is why each STM algorithm is more effective in some specific workloads than others.

We design and implement RInval with the goal of minimizing the effect of most previously mentioned overheads. More in detail, we alleviate the effect of *i)* the locking overhead, *ii)* the tradeoff between validation and commit overheads, and *iii)* the abort overhead. The overhead of meta-data logging usually cannot be avoided in lazy algorithms. For locking, we used the same idea of RTC by executing commit routines in a dedicated server core. Validation and commit are improved by using invalidation outside, and in parallel with, the main commit routine. Finally, we use a simple contention manager to reduce the abort overhead.

9.2 Remote Invalidation

As described in Section 9.1, *Remote Invalidation* reduces the overhead of the transaction critical path. To simplify the presentation, we describe the idea incrementally, by presenting three versions of RInval[2]. In the first version, called RInval-V1, we show how spin locks are replaced by the more efficient remote core locks. Then, in RInval-V2, we show how commit and invalidation are parallelized. Finally, in RInval-V3, we further optimize the algorithm by allowing the commit-server to start a new commit routine before invalidation-servers finish their work.

9.2.1 Version 1: Managing the locking overhead

RInval-V1 uses the same idea of RTC: commit routines are executed remotely to replace spin locks. Figure 9.4 shows how RInval-V1 works. When a client reaches a commit phase, it sends a commit request to the commit-server by modifying a local *request_state* variable to be PENDING. The client then keeps spinning on *request_state* until it is changed by the server to be either ABORTED or COMMITTED. This way, each transaction spins on its own variable instead of competing with other transactions on a shared lock.

Figure 9.5 shows the structure of the cache-aligned requests array. In addition to *request_state*, the commit-server only needs to know two values: *tx_status*, which is used to check if the transaction has been invalidated in an earlier step, and *write_set*, which is used for publishing writes on shared memory and for invalidation. In addition, padding bits are added to cache-align the request.

Since we use a coarse-grained approach, only one commit-server is needed. Adding more than one commit-server will cause several overheads: *i)* the design will become more complex, *ii)* more cores have to be dedicated for servers, *iii)* more CAS operations must be added to

[2]We only present the basic idea in the pseudo code given in this section. The source code provides the full implementation details.

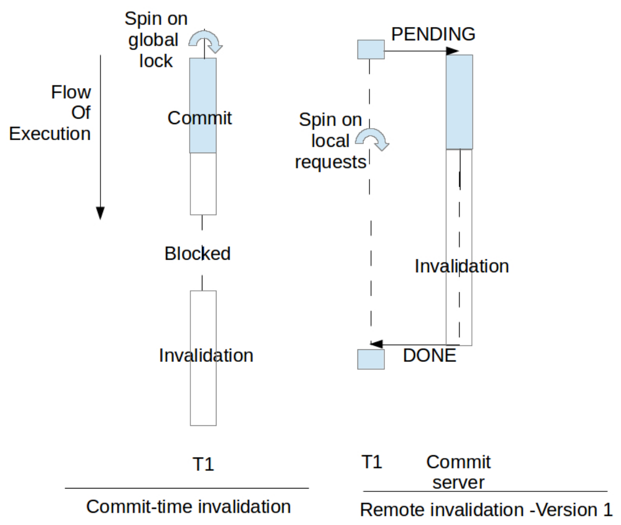


Figure 9.4: Flow of commit execution in both InvalSTM and RInval-V1

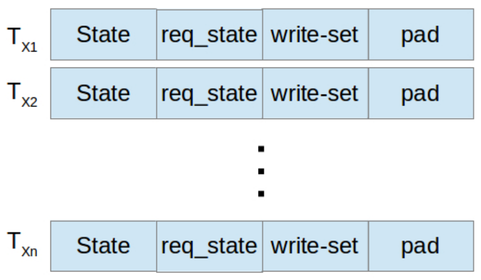


Figure 9.5: RInval's Cache-aligned requests array

synchronize the servers, and *iv)* cache misses may occur among servers. Since we minimize the work done by the commit-server, the overhead of serializing commit on one server is expected to be less than these overheads.

Algorithm 25 shows the pseudo code of RInval-V1. We assume that instructions are executed in the same order as shown (i.e., sequential consistency is assumed). We ensure this in our C/C++ implementation by using memory fence instructions when necessary (to prevent out-of-order execution), and by using volatile variables when necessary (to prevent compiler re-ordering). This version modifies the InvalSTM algorithm shown in Section 2.1.2.

The read procedure is the same in both InvalSTM and RInval, because we only shift execution of commit from the application thread to the commit-server. In the commit procedure, if the transaction is read-only, the commit routine consists of only clearing the local variables. In write transactions, the client transaction checks whether it was invalidated by an earlier commit routine (line 5). If validation succeeds, the client changes its state to PENDING (line 7). The client then loops until the commit-server handles its commit request and changes

Algorithm 25 Remote Invalidation - Version 1

```
1: procedure CLIENT COMMIT
2:     if $read\_only$ then
3:         ...
4:     else
5:         if $tx\_status$ = INVALIDATED then
6:             TxAbort()
7:         $request\_state$ = PENDING
8:         loop while $request\_state \notin$ (COMMITTED, ABORTED)
9: end procedure

10: procedure COMMIT-SERVER LOOP
11:     while $true$ do
12:         for $i \leftarrow 1, num\_transactions$ do
13:             $req \leftarrow requests\_array[i]$
14:             if $req.request\_state$ = PENDING then
15:                 if $req.tx\_status$ = INVALIDATED then
16:                     $req.request\_state$ = ABORTED
17:                 else
18:                     $timestamp$++
19:                     for All in-flight transactions $t$ do
20:                         if $me.write\_bf$ intersects $t.read\_bf$ then
21:                             $t.tx\_status$ = INVALIDATED
22:                     WriteInMemory($req.writes$)
23:                     $timestamp$++
24:                     $req.request\_state$ = COMMITTED
25: end procedure
```

the state to either COMMITTED or ABORTED (line 8). The client will either commit or roll-back according to the reply.

On the server side, the commit-server keeps looping on client requests until it reaches a PENDING request (line 14). The server then checks the client's *request_state* to see if the client has been invalidated (line 15). This check has to be repeated at the server, because some commit routines may take place after sending the commit request and before the commit-server handles that request. If validation fails, the server changes the state to ABORTED and continues searching for another request. If validation succeeds, it starts the commit operation (like InvalSTM). At this point, there are two main differences between InvalSTM and RInval-V1. First, incrementing the timestamp does not use the CAS operation (line 18), because only the main server changes the timestamp. Second, the server checks *request_state* before increasing the timestamp (line 15), and not after it, like in InvalSTM, which saves the overhead of increasing the shared timestamp for a doomed transaction. Since only the commit-server can invalidate transactions, there is no need to check *request_state* again after increasing the timestamp.

9.2.2 Version 2: Managing the tradeoff between validation and commit

In RInval-V1, we minimized the overhead of locking on the critical path of transactions. However, invalidation is still executed in the same routine of commit (in serial order with commit itself). RInval-V2 solves this problem by dedicating more servers to execute invalidation in parallel with commit. Unlike the commit-server, there can be more than one invalidation-server, because their procedures are independent. Each invalidation-server is responsible for invalidating a subset of the running transactions. The only data that needs to be transferred from the commit-server to an invalidation-server is the client's write-set. Figure 9.6 shows RInval-V2 with one commit-server and two invalidation-servers. When the commit-server selects a new commit request, it sends the write bloom filter of that request to the invalidation-servers, and then starts execution. When the commit-server finishes, it waits for the response from all invalidation-servers, and then proceeds to search for the new commit request.

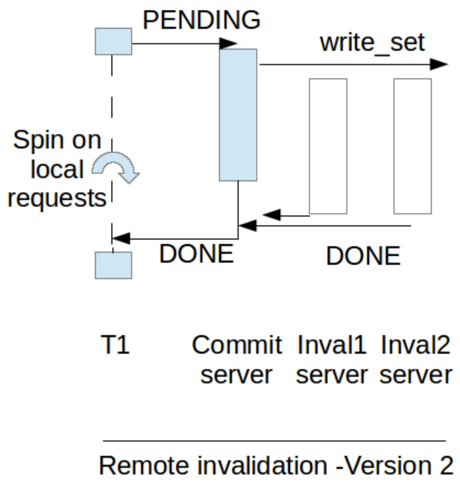


Figure 9.6: Flow of commit execution in RInval-V2

Selecting the number of invalidation-servers involves a trade-off. According to Amdahl's law, concurrency decreases as the number of parallel executions increases. At some point, adding more invalidation-servers may not have a noticeable impact on performance. At the same time, increasing the number of invalidation-servers requires dedicating more cores for servers, and adds the overhead of servers communication. In our experiments, we found that on a 64-core machine, it is sufficient to use 4 to 8 invalidation-servers to achieve the maximum performance.

Adding invalidation-servers does not change the fact that no CAS operations are needed. It also ensures that all communication messages (either between the commit-server and the clients, or between the commit-server and the invalidation-servers) are sent/received using cache-aligned requests. Thus, RInval-V2 inherits the benefits of optimized locking and parallelizing commit-invalidation routines.

Algorithm 26 Remote Invalidation - Version 2

```
 1: procedure COMMIT-SERVER LOOP
 2:     while $true$ do
 3:         for $i \leftarrow 1, num\_transactions$ do
 4:             $req \leftarrow requests\_array[i]$
 5:             if $req.request\_state$ = PENDING then
 6:                 for $i \leftarrow 1, num\_invalidators$ do
 7:                     while $timestamp > inval\_timestamp$ do
 8:                         LOOP
 9:                 if $req.tx\_status$ = INVALIDATED then
10:                     $req.request\_state$ = ABORTED
11:                 else
12:                     $commit\_bf \leftarrow req.write\_bf$
13:                     $timestamp$++
14:                     WriteInMemory($req.writes$)
15:                     $timestamp$++
16:                     $req.request\_state$ = COMMITTED
17: end procedure

18: procedure INVALIDATION-SERVER LOOP
19:     while $true$ do
20:         if $timestamp > inval\_timestamp$ then
21:             for All in-flight transactions $t$ in my set do
22:                 if $commit\_bf$ intersects $t.read\_bf$ then
23:                     $t.tx\_status$ = INVALIDATED
24:             $inval\_timestamp$ += 2
25: end procedure

26: procedure CLIENT READ
27:     ...
28:     if $x1 = timestamp$ and $timestamp = my\_inval\_timestamp$ then
29:         ...
30: end procedure
```

Algorithm 26 shows RInval-V2's pseudo code. The client's commit procedure is the same as in RInval-V1, so we skip it for brevity. Each invalidation-server has its local timestamp, which must be synchronized with the commit-server. The commit-server checks that the timestamp of all invalidation-servers is greater than or equal to the global timestamp (line 7). It then copies the write bloom filter of the request into a shared *commit_bf* variable to be accessed by the invalidation-servers (line 12).

The remaining part of RInval-V2 is the same as in RInval-V1, except that the commit-server does not make any invalidation. If an invalidation-server finds that its local timestamp has become less than the global timestamp (line 20), it means that the commit-server has started handling a new commit request. Thus, it checks a subset of the running transactions (which are evenly assigned to servers) to invalidate them if necessary (lines 21-23). Finally, it increments its local timestamp by 2 to catch up with the commit-server's timestamp (line 24). It is worth noting that the invalidation-server's timestamp may be greater than the commit-server's global timestamp, depending upon who will finish first.

The client validation is different from RInval-V1. The clients have to check if their invalidation-servers' timestamps are up-to-date (line 28). The invalidation-servers' timestamps are always increased by 2. This means that when they are equal to the global timestamp, it is guaranteed that the commit-server is idle (because its timestamp is even).

9.2.3 Version 3: Accelerating Commit

In RInval-V2, commit and invalidation are efficiently executed in parallel. However, in order to be able to select a new commit request, the commit-server must wait for all invalidation-servers to finish their execution. This part is optimized in RInval-V3. Basically, if there is a new commit request whose invalidation-server has finished its work, then the commit-server can safely execute its commit routine without waiting for the completion of the other invalidation-servers. RInval-V3 exploits this idea, and thereby allows the commit-server to be n steps ahead of the invalidation-servers (excluding the invalidation-server of the new commit request).

Algorithm 27 Remote Invalidation - Version 3

```
1:  procedure COMMIT-SERVER LOOP
2:      if $req.request\_state$ = PENDING and $req.inval\_timestamp \geq timestamp$ then
3:          ...
4:      ...
5:      while $timestamp > inval\_timestamp + num\_steps\_ahead$ do
6:          LOOP
7:      ...
8:      $commit\_bf[my\_index++] \leftarrow req.write\_bf$
9:      ...
10: end procedure

11: procedure INVALIDATION-SERVER LOOP
12:     ...
13:     if $commit\_bf[my\_index++]$ intersects $t.read\_bf$ then
14:         ...
15: end procedure
```

Algorithm 27 shows how RInval-V3 makes few modifications to RInval-V2 to achieve its goal. In line 2, the commit-server has to select an up-to-date request, by checking that the timestamp of the request's invalidation-server equals the global timestamp. The commit-server can start accessing this request as early as when it is n steps ahead of the other invalidation-servers (line 5). All bloom filters of the requests that do not finish invalidation are saved in an array (instead of one variable as in RInval-V2). This array is accessed by each server using a local index (lines 8 and 13). This index is changed after each operation to keep pointing to the correct bloom filter.

It is worth noting that, in the normal case, all invalidation-servers will finish almost in the same time, as the clients are evenly assigned to the invalidation-servers, and the invalidation process takes almost constant time (because it uses bloom filters). However, RInval-V3 is more robust against the special cases in which one invalidation-server may be delayed for some reason (e.g., OS scheduling, paging delay). In these cases, RInval-V3 allows the commit-server to proceed with the other transactions whose invalidation-servers are not blocked.

9.2.4 Other Overheads

In the three versions of RInval, we discussed how we alleviate the overhead of spin locking, validation, and commit. As discussed in Section 9.1, there are two more overheads that affect the critical path of transactions. The first is logging, which cannot be avoided as we use a lazy approach. This issue is not just limited to our algorithm. Storing reads and writes in local read-sets and write-sets, respectively, is necessary for validating transaction consistency. The second overhead is due to abort. Unlike InvalSTM, we prevent the contention manager from aborting or delaying the committing transaction even if it conflicts with many running transactions. This is because of two reasons. First, it enables finishing the invalidation as early as possible (in parallel with the commit routine), which makes the abort/retry procedure faster. Second, we shorten the time needed to complete the contention manager's work, which by itself is an overhead added to the servers' overhead, especially for the common case (in which writers invalidate readers).

9.2.5 Correctness and Features

RInval guarantees opacity in the same way other coarse-grained locking algorithms do, such as NOrec [25] and InvalSTM [40]. Both reads and writes are guaranteed to be consistent because of lazy commit and global commit-time locking. Before each new read, the transaction check that *i)* it has not been invalidated in an earlier step, and *ii)* no other transaction is currently executing its commit phase. Writes are delayed to commit time, which are then serialized on commit-servers. The only special case is that of RInval-V3, which allows the commit-server to be several steps ahead of invalidation. However, opacity is not violated here, because this step-ahead is only allowed for transactions whose servers have finished invalidation.

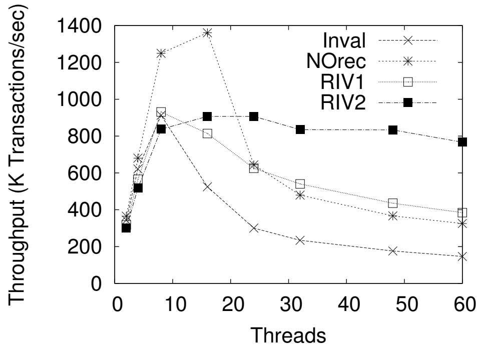
RInval also inherits all of the advantages of coarse-grained locking STM algorithms, including simple global locking, minimal meta-data usage, privatization safety [97], and easy integration with hardware transactions [91]. Hardware transactions need only synchronize with the commit-server, because it is the only thread that writes to shared memory.

9.3 Evaluation

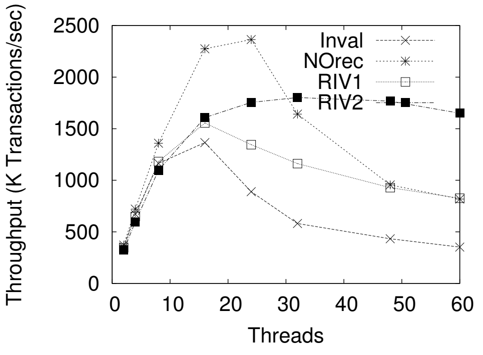
We implemented RInval in C/C++ (compiled with gcc 4.6) and ported to the RSTM framework [73] (compiled using default configurations) to be tested using its interface. Our experiments were performed on a 64-core AMD Opteron machine (128GB RAM, 2.2 GHz).

To assess RInval, we compared its performance against other coarse-grained STM algorithms, which have the characteristics as RInval, like minimal meta-data, easy integration with HTM, and privatization safety. We compared RInval with InvalSTM [40], the corresponding non-remote invalidation-based algorithm, and NOrec [25], the corresponding validation-based algorithm. For both algorithms, we used their implementation in RSTM with the default configuration. We present the results of both RInval-V1 and RInval-V2 with 4 invalidation-servers. For clarity, we withheld the results of RInval-V3 as it resulted very close to RInval-V2. This is expected because we dedicate separate cores for invalidation-servers, which means that the probability of blocking servers is minimal (recall that blocking servers is the only case that differentiate RInval-V2 from RInval-V3)

We show results in red-black tree micro-benchmark and the STAMP benchmark [81]. In these experiments, we show how RInval solves the problem of InvalSTM and becomes better than NOrec in most of the cases. All of the data points shown are averaged over 5 runs.



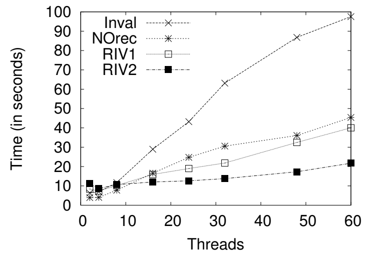
(a) 50% reads



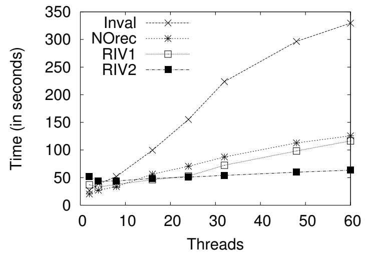
(b) 80% reads

Figure 9.7: Throughput (K Transactions per second) on red-black tree with 64K elements

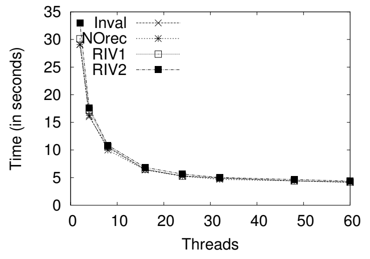
Red-Black Tree. Figure 9.7 shows the throughput of RInval and its competitors for a red-black tree with 64K nodes and a delay of 100 no-ops between transactions, for two different workloads (percentage of reads is 50% and 80%, respectively). Both workloads execute a series of red-black tree operations, one per transaction, in one second, and compute the overall throughput. In both cases, when contention is low (less than 16 threads), NOrec performs better than all other algorithms, which is expected because invalidation benefits take place only in higher contention levels. However, RInval-V1 and RInval-V2 are closer to NOrec than InvalSTM, even in these low contention cases. As contention increases (more than 16 threads), performance of both NOrec and InvalSTM degrades notably, while both RInval-V1 and RInval-V2 sustain their performance. This is mainly because NOrec and InvalSTM use spin locks and suffer from massive cache misses and CAS operations, while RInval isolates commit and invalidation in server cores and uses cache-aligned communication. RInval-V2 performs even better than RInval-V1 because it separates and parallelizes commit and in-



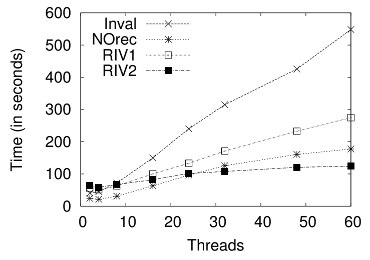
(a) kmeans



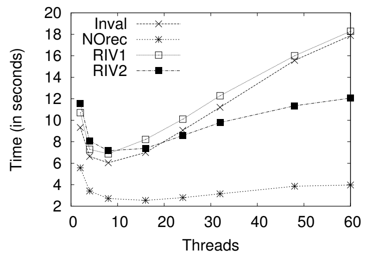
(b) ssca2



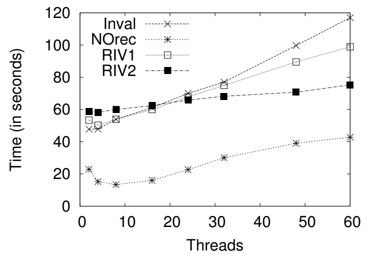
(c) labyrinth



(d) intruder



(e) genome



(f) vacation

Figure 9.8: Execution time on STAMP benchmark

validation routines. RInval-V2 enhances performance as much as 2x better than NOrec and 4x better than InvalSTM.

STAMP. Figure 9.8 shows the results of the STAMP benchmark, which represents more realistic workloads. In three benchmarks (*kmeans*, *ssca*2, and *intruder*), RInval-V2 has the best performance starting from 24 threads, up to an order of magnitude better than InvalSTM and 2x better than NOrec. These results confirm how RInval solves the problem of serializing commit and invalidation, which we showed in Figure 9.3. In *genome* and *vacation*, NOrec is better than all invalidation algorithms. This is mainly because they are read-intensive benchmarks, as we also showed in Figure 9.3. However, RInval is still better and closer to NOrec than InvalSTM. For future work, we can make further enhancements to make these specific cases even better. One of these enhancements is to bias the contention manager to readers, and allow it to abort the committing transaction if it is conflicting with many readers (instead of the classical *winning commit* mechanism, currently used). In *labyrinth*, all algorithms perform the same, which confirms the claim made in Section 9.1, because their main overhead is non-transactional. We did not show *bayes* as it has a large variance in its results.

9.4 Summary

There are many parameters – e.g., spin locking, validation, commit, abort – that affect the critical execution path of memory transactions and thereby transaction performance. Importantly, these parameters interfere with each other. Therefore, reducing the negative effect of one parameter (e.g., validation) may increase the negative effect of another (i.e., commit), resulting in an overall degradation in performance for some workloads.

In this chapter we presented RInval, an STM implementation showing that it is possible to mitigate the effect of all of the critical path overheads. RInval dedicates server cores to execute both commit and invalidation in parallel, and replaces all spin locks and CAS operations with server-client communication using cache-aligned messages. This optimizes lock acquisition, incremental validation, and commit/abort execution, which are the most important overheads in the critical path of memory transactions.

Chapter 10

Modeling Transactional Data Structures

The last two decades witnessed many efficient designs of concurrent data structures. A large set of them shares a common design principle: each operation is split into a *read-only traversal* phase, which scans the data structure without locking or monitoring, and a *read-write commit* phase, which atomically validates the output of the *traversal* phase and applies the needed modifications to the data structure. As we showed in Chapter 3, our OTB methodology, along with other different approaches [3, 101], extended those designs for allowing the composition of multiple operations into atomic transactions by building a single *read-only* phase and a single *update* phase for the whole transaction. As a result, we observed a set of data structures that are *optimistic* and composable. The former because they defer any locking and/or monitoring to the *comit* phase; the latter because they allow atomic executions of multiple operations. Relying on our discussion about Optimistic Semantic Synchronization (OSS) in Section 3.4, we name the set of data structures (either concurrent or transactional) that have the aforementioned characteristics as *OSS data structures.*

Although all the *OSS data structures* are designed similarly, the literature lacks of a unified model to reason about the correctness of those designs. In this chapter we made a step towards filling this gap by leveraging a recent approach that models data structures with concurrent readers and just a single writer (called single writer multiple reader, or SWMR).

10.1 Background and Definitions

In this section we illustrate the background of this paper, which is composed of some basic definitions and an overview of the *SWMR* model [70]. Throughout the paper, we use similar terminologies as [70] in order to allow an easy comparison with other existing models.

We define a data structure as a set of shared variables $X = \{x_1, x_2, \ldots x_i, \ldots\}$ where operations can be invoked on. An operation execution[1] O, is a sequence of $Steps_O = s_O^1 \cdot s_O^2 \cdot \ldots \cdot s_O^n$, where: s_O^1 is the invocation of the operation, i.e., $invoke_O$, and s_O^n is the operation's *return*, i.e., $return_O(v_{ret})$. A dummy $return(void)$ step is added for any operation that does not have an explicit return value. Any other step is either $read_O(x_i)$ or $write_O(x_i, v_w)$ (those steps comply with their common meaning). A *transaction* T is a sequence of operations surrounded by a *begin* and *commit/abort* steps ($T = begin \cdot O_1 \cdot O_2 \cdot \ldots \cdot O_k \cdot commit/abort$). Steps are assumed to be executed atomically. The *size* of a transaction is the number of operations it executes. An operation is called *read-only* if it does not execute any *write* step; otherwise it is called *update*. Consequently, a *transaction* is *read-only* if all its operations are *read-only*. A sequential execution of a data structure *ds* is a sequence of non-interleaving operations/transactions on *ds*. A *concurrent execution* μ is a sequence of interleaved steps of different operations. A *transactional execution* μ is a sequence of interleaved steps of different transactions.

The history H of a concurrent execution μ on a data structure *ds*, $H|\mu$, is the subsequence of μ with only the *invoke* and the *return* steps. In the case of a transactional executions, $H|\mu$ also contains the begin, commit and abort steps of the transactions. A *pending* operation (transaction) in H is an operation (transaction) with no *return*(*commit*) step. *complete(H)* is the sub-history of H with no pending operations (respectively transactions). A history H is sequential if no two operations (respectively transactions) in H are concurrent. We say that two operations (respectively transactions) are concurrent if the return step (respectively either the commit or the abort step) of one does not precede the invoke (respectively the begin) step of the other one.

Any data structure *ds* has a *sequential specification*, which corresponds to the set of all the allowed sequential histories on *ds*. A history H is *linearizable* [60] if it can be extended (by appending zero or more *return (respectively commit)* steps) to some history H' that is equivalent (i.e., has the same operations, transactions, and *return/commit/abort* steps) to a legal (i.e., satisfies the sequential specification) sequential history S such that non-interleaving operations (and transactions) in H' appear in the same order in S.

The *shared state* s of the data structure is defined at any time by the values of its shared variables, and it is selected from a set of shared states $\mathcal{S}$. Each operation (respectively transaction) has a *local state* l, which is defined by the values of its local variables, that is selected from a set of local states $\mathcal{L}_{op}(\mathcal{L}_T)$. The sets $\mathcal{S}$, $\mathcal{L}_{op}$ and $\mathcal{L}_T$ contain initial states $\mathcal{S}_0$, $\bot_{op}$, and $\bot_T$, respectively (we call both the local states $\bot$ for simplicity). A *step* in the execution of each operation (respectively transaction) represents a transition function on $\mathcal{S}$ and $\mathcal{L}_{op}$ (respectively $\mathcal{L}_T$) that changes the *shared state* of the data structure and the *local state* of the operation (respectively transaction) from $\langle l, s\rangle$ to $\langle l', s'\rangle$. At any point of a concurrent (respectively transactional) execution μ, if we have l pending operations

[1] In this paper we also use the term *operation (transaction)* to indicate *an operation (a transaction) execution.*

(transactions), we will have l local states (one for each operation/transaction) and one shared state s.

Given a data structure ds and an operation O (a transaction T) in a concurrent (respectively transactional) execution μ on ds, we define $\textit{pre-state}_O$ ($\textit{pre-state}_T$) as the shared state of ds right before $\textit{invoke}_O$ (respectively $\textit{begin}_T$), and $\textit{post-state}_O$ ($\textit{post-state}_T$) as the shared state of ds right after $\textit{return}_O(v_{ret})$ (respectively $\textit{commit}_T$). We also say that a *shared state* is *sequentially reachable* if it can be reached in some sequential execution of a data structure.

The *single writer multiple reader (SWMR)* model assumes *concurrent executions* on a data structure ds where the steps of two *update* operations in any concurrent execution μ on ds do not interleave (conversely, a *multiple writer multiple reader (MWMR)* model is the one that allows such an interleaving). In the following we report the definition of *base condition* and *base point* as defined in the *SWMR* model of [70]. Note that, unlike in [70], in our models those definitions apply to all operation/transactions and not only to read-only operations.

- *base condition*: Given a local state l of an operation O on a data structure ds, a base condition ϕ for l is a predicate over the shared state of ds where every sequential execution of O starting from a shared state s such that $\phi(s) = \textit{true}$, reaches l. A base condition for a step s^i (named also ϕ^i) is the base condition for the local state right before the execution of s^i.

- *base point*: An execution of a step s^i, with a local state l, in a concurrent execution μ has a base point if there is a sequentially reachable *post-state* s, such that the base condition $\phi^i(s)$ holds.

The combination of the two definitions makes an interesting conclusion: if an execution of a step s^i in an operation O has a base point in a concurrent execution μ, this means that there is a sequentially reachable *post-state* from which O can start and reach s^i with the same local state l. That also means that the execution of s^i in μ would have been the same as performed in a sequential execution. Accordingly, if every step in every concurrent execution of ds has a base point, then we say that ds is *valid*. Informally, that means that ds is never subject to "bad behaviors" (e.g., division by zero or null-pointer accesses).

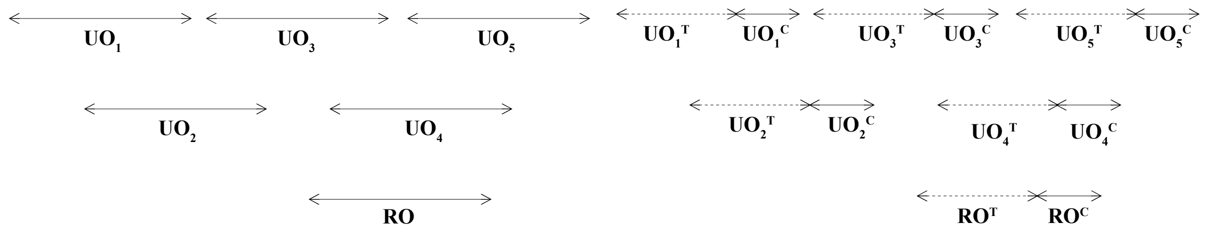
In addition to the above definitions, the *SWMR* model [70] names a data structure ds as *regular* if for each history H, the sub-history composed of all write operations in H enriched with one read-only operation (if any) in H is linearizable. The *SWMR* model ensures regularity by restricting the candidate *base points* for the *return* steps of every *read-only* operation ro to be the *post-state* of either an *update* operation executed concurrently with ro in μ or of the last *update* operation that ended before ro's *invoke* step in μ (those candidate *base points* are called *regularity base points*).

10.2 The Single Writer Commit (SWC) Model

As mentioned before, in our models each operation is split into *read-only traversal* phase and *read-write commit* phase. This representation is general enough to cover also those operations with either an empty traversal (i.e., operations whose first step is a write) or an empty commit phase (i.e., read-only operations).

In this section we present the Single Writer Commit (SWC) model, a *MWMR* model in which both *read-only* and *update* operations run concurrently with the restriction that only the *commit* phases are atomically executed with the *Single Lock Atomicity (SLA)* semantics (i.e., as if they are executed sequentially). For the sake of simplifying the presentation, we first introduce this model by assuming that the commit phases are protected by a single global lock. Then, in Section 10.2.1 we discuss the case of concurrent commits.

Figure 10.1 shows an example of this case with five *update* operations, $uo_1, ..., uo_5$, and one *read-only* operation ro. In this example, the commit phases of all the *update* operations do not interleave, even if the operations themselves interleave. The *read-only* operation ro is concurrent with uo_3, uo_4, and uo_5. In particular, it interleaves with the commit phases of uo_3 and uo_4, while its *commit* phase only interleaves with uo_4.



(a) Multiple Writers (b) Single Writer Commit

Figure 10.1: An example of a *MWMR* concurrent execution (a) that can be executed using our model by converting it to a *single writer commit* scenario (b).

Algorithm 28 shows a practical (and simple) data structure implementation under the *SWC* model: a linked-list with three operations *readLast*, *insertLast*, and *removeLast* (which gives the semantics of stacks). The head of the list is assumed to be constant (i.e., the widely used sentinel node). We defer to Section 10.4 the discussion about more complex and practical cases. Although all the executions of *readLast* (respectively *insertLast*) are *read-only* (respectively *update*), some execution of *removeLast* (those that return at line 27) are *read-only* and some others (those that return at line 37) are *update*. Unlike the *SWMR* model, *SWC* does not categorize operations in a concurrent execution μ as *read-only* or *update* a priori, but rather it assigns the operation's type considering its actual execution in μ, which therefore increases the level of concurrency. The *SWC* model considers *executions* rather than *operations* as its building blocks. Accordingly, in Algorithm 28, *removeLast* is not treated as an *update* operation when the linked-list is empty.

Algorithm 28 A linked list with three operations implemented under the *SWC* model.

```
1: procedure READLAST
2:     $last \leftarrow \perp$
3:     $next \leftarrow$ read($head.next$)                    ▷ $\phi_1 : true$
4:     while $next \neq \perp$ do
5:         $last \leftarrow next$
6:         $next \leftarrow$ read($last.next$)            ▷ $\phi_2 : head \overset{*}{\Rightarrow} last$
7:     return($last$)                                ▷ $\phi_3 : head \overset{*}{\Rightarrow} last$
8: end procedure

9: procedure INSERTLAST($n$)
10:    $last \leftarrow \perp$
11:    $next \leftarrow$ read($head.next$)                    ▷ $\phi_4 : true$
12:    while $next \neq \perp$ do
13:        $last \leftarrow next$
14:        $next \leftarrow$ read($last.next$)            ▷ $\phi_5 : head \overset{*}{\Rightarrow} last$
15:    lockAcquire($gl$)
                                                     ▷ $\phi_6 : head \overset{*}{\Rightarrow} last$
16:    if read($last.next$) $\neq \perp$ then
17:        lockRelease($gl$)
18:        go to 10
19:    write($last.next$, $n$)
20:    lockRelease($gl$)
21: end procedure

22: procedure REMOVELAST
23:    $last \leftarrow \perp$
24:    $secondlast =$ read($head$)                          ▷ $\phi_7 : true$
25:    $next \leftarrow$ read($head.next$)                    ▷ $\phi_8 : true$
26:    if $next = \perp$ then
27:        return                                    ▷ $\phi_9 : head.next = \perp$
28:    while $next \neq \perp$ do
29:        $secondLast = last$
30:        $last \leftarrow next$
31:        $next \leftarrow$ read($last.next$)           ▷ $\phi_{10} : head \overset{*}{\Rightarrow} last$
32:    lockAcquire($gl$)
                                                     ▷ $\phi_{11} : head \overset{*}{\Rightarrow} last$
33:    if read($last.next$) $\neq \perp$ then
34:        lockRelease($gl$)
35:        go to 23
36:    write($secondlast.next$, $\perp$)
37:    lockRelease($gl$)
38: end procedure
```

Figure 10.2(a) shows how we model a typical *OSS data structure* operation. Any operation O is split into two sequences of steps: $O^T = s^1 \cdot ... \cdot s^m$; and $O^C = s^{m+1} \cdot ... \cdot s^n$. The sequence O^T represents the *traversal* phase, which does not contain any *write* step. The sequence O^C represents the *commit* phase, which always ends with $return_O(v_{ret})$ and can contain both *read* and *write* steps. Given that a data structure under the *SWC* model allows concurrent *traversal* phases and a single *commit* phase at a time, the transitions from the *shared traversal* phase to the *exclusive commit* phase and vice versa are represented by two auxiliary steps S' and S'' (e.g., they can be an acquisition/release of a global lock as in Algorithm 28). We do not assume the presence of such a transition in *read-only* operations, thus, in those cases, S' and S'' are just dummy steps that do nothing. Excluding the auxiliary steps, the commit phase of a read-only operation O is $O^C = return_O(v_{ret})$.

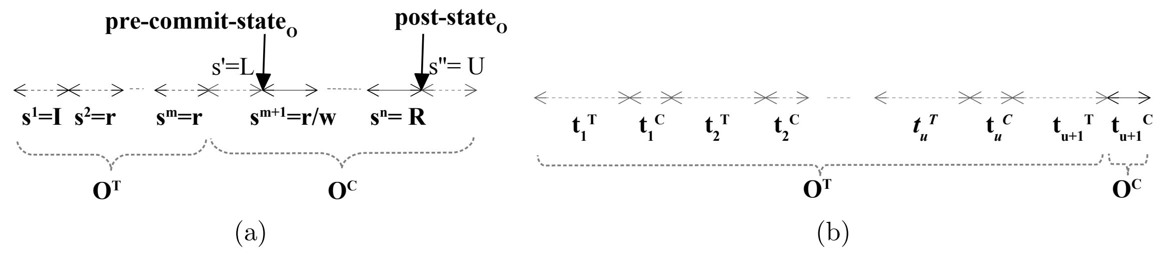


Figure 10.2: *a)* Splitting the operation to support concurrent MWMR execution with single writer commit (SWC). O^T is the *traversal* phase; O^C is the *commit* phase. I:*invoke*, r:*read*, w:*write*, R:*return*, L:*lock*, U:*unlock*. *b)* Unsuccessful trials are part of the overall *traversal* phase in our model.

In practice, *OSS data structures* usually start the *commit* phase by a validation mechanism to ensure that the output of the traversal phase remains valid until the transition to the exclusive *commit* mode; otherwise the *traversal* phase is re-executed. That is why it is important to include this re-execution mechanism in our model. To do so, we define for each operation O on a data structure *ds* a variable u that represents the number of *unsuccessful trials* ($u \in \{0, 1, ..., \infty\}$)[2]. The value of u is determined according to the design of *ds* and the concurrent execution μ that includes O. Every unsuccessful trial resets the local state of the operation to the initial $\perp$ state before starting the next trial. The commit phases of all the unsuccessful trials are clearly not allowed to write on the shared memory because of their inconsistent local state. As shown in Figure 10.2(b), the *traversal* phase of the operation O includes all those unsuccessful trials $t_1 \cdot t_2 \cdot ... \cdot t_u$, and the *commit* phase of O is only the successful *commit* phase of the last trial (${t_{u+1}}^C$).

In Algorithm 28, the *commit* phase of *readLast* is always successful (in fact, the operation itself is wait-free [56]). Thus, it is easy to identify the *traversal* phase (the whole execution before line 7), and the *commit* phase (the *return* step at line 7). Identifying *insertLast*'s phases is more complicated because it may have unsuccessful *commit* phases. According to our definitions, the *traversal* phase of an operation O with u unsuccessful trials is a concatenation of u executions of lines (2 - 18) in which the condition of line 16 is false, followed by one execution until right before line 15. The *commit* phase is formed by lines (15 - 20) in which the condition of line 16 is true. The phases of the read-only (respectively update) executions of *removeLast* are determined similar to *readLast* (respectively *insertLast*).

The definitions of *base conditions*, *base points*, and *validity* are similar to the $SWMR$ model, but in the SWC model they are defined for both *read-only* and *update* operations. However, the definitions of *regularity base point* and *regularity* need to be refined. First, as SWC is a $MWMR$ model, we have more than one definition of regularity on registers [92] and there is no prior work that discusses which of them can be applied to data structures. Second, putting momentarily this point aside (we will take into account this point later in Section 10.4), the weakest definition of *regularity* in [92] entails that, for a history H, every sub-history that contains all the writes plus one read is linearizable. In the $SWMR$ model, *update* operations are linearizable because they are executed sequentially. However, in the SWC model, this is not trivially guaranteed because an *update* operation may be invalidated before performing its exclusive *commit* phase. To avoid such an issue, in our model we first define a new state for each update operation *uo*, called *pre-commit-state*$_{uo}$, which represents the local state of *uo* after the auxiliary step s' and before the first *real* step in the *commit* phase, s^{m+1}. Then, we guarantee the linearization of the *update* operations as follows.

Definition 1. ***(Executions under*** $\boldsymbol{SWC}$***)*** *In a concurrent execution μ with k update operations whose commit phases are sequential, those k operations are totally ordered according to the order of their commit phases $\{u_1 \prec_c u_2 \prec_c ... \prec_c u_k\}$. A dummy u_0 operation is*

[2]If for an operation O it is possible to have an execution with $u = \infty$, this (informally) entails that the operation is not wait-free.

added such that post-state$_{u_0}$ is $\mathcal{S}_0$. A concurrent execution μ is under the SWC model if update operations have sequential commit phases, and for every update operation u_i in μ, pre-commit-state$_{u_i}$ observes post-state$_{u_{i-1}}$.

From now on, we focus on those executions under the *SWC* model according to Definition 1. Theorem 5 shows that in those executions, *update* operations are linearizable.

Theorem 5. *Given a concurrent execution μ with k completed update operations, and the corresponding history of those k operations $H_k|\mu$, if μ is under SWC model, all the post-states of the k operations are sequentially reachable, and $H_k|\mu$ is linearizable.*

Proof. Let us consider an execution μ under the *SWC* model with k completed update operations such that they are ordered according to the order of their *commit* phases (i.e., $u_1 \prec_c u_2 \prec_c \dots \prec_c u_k$) by Definition 1. Let us also consider the history $H_k|\mu$ that contains the *invocation* and *return* steps of those k operations. We prove that all the *post-states* of the k operations are sequentially reachable by induction.

As base step, we prove that the *post-state* of the first *update* operation committing in μ, i.e., u_1, is sequentially reachable. This is guaranteed because u_1 is the first writer by construction of μ, and no any other operation can concurrently apply write steps when u_1 is executing its commit phase given Definition 1 of *SWC*. Therefore u_1's base point is the initial state.

As inductive step, we suppose that the *post-states* of the operations $u_1 \prec_c u_2 \prec_c \dots \prec_c u_{i-1}$, with $1 < i \leq k$, are sequentially reachable, and we prove that also the *post-state* of u_i is sequentially reachable. According to Definition 1, *pre-commit*$_{u_i}$ observes *post-state*$_{u_{i-1}}$ which means that u_i starts its commit phase observing a *sequentially reachable* shared state. As u_i^C is executed in isolation from concurrent writers (by Definition 1), it is able to generate a *sequentially reachable post-state* right before it reaches the auxiliary step s''. Thus, *post-state*$_{u_i}$ is *sequentially reachable.*

We now prove that $H_k|\mu$ is linearizable. By construction of μ, $H_k|\mu$ contains only completed operations, thus $complete(H_k|\mu) = H_k|\mu$. Then, consider a sequential history S that contains the k update operations in the order of $\prec_c$. Clearly, $\prec_{H_k|\mu} \subseteq \prec_S$ because if $u_i \prec_{H_k|\mu} u_j$, the return step of u_i precedes the invoke step of u_j. Therefore $u_i \prec_c u_j$, and thus $u_i \prec_S u_j$. Then, for each operation u_i in S, *pre-commit*$_{u_i}$ also observes *post-state*$_{u_{i-1}}$, which infers that the operations have the same return values in both S and $H_k|\mu$, i.e., S and $H_k|\mu$ are equivalent. Hence, $H_k|\mu$ is linearizable.

□

The intuition of the proof is that the operation that commits first trivially produces a *sequentially reachable* state. Based on Definition 1, at commit time each operation observes the *post-state* of the operation right before it. Then, by induction, all operations produce *sequentially reachable* states. Therefore, *update* operations are linearized according to the order of their *commit* phases.

Based on Theorem 5, in any concurrent execution μ with k completed *update* operations, we can identify $k+1$ *sequentially reachable* shared states that would be the candidate *base points* for each step in μ. Those points are the *post-states* of the k completed *update* operations, in addition to the initial state $\mathcal{S}$. Next, we refine the definition of *regularity base points* as follows:

Definition 2. ***(Regularity base points under SWC)*** *A base point bp of a step s^i in a read-only operation ro of a concurrent execution μ under the SWC model is a regularity base point if bp is the post-state of either an update operation whose commit phase is executed concurrently with ro in μ or of the update operation whose commit phase is the last one completed before ro's invoke step in μ (the initial state is the default).*

This definition simply restricts the candidate *regularity base points* of any *read-only* operation to be the *post-state* of the operations with interleaving *commit* phases rather than those of the interleaving *update* operations. For example, in Figure 10.1 the *post-state* of uo_3 and uo_4 are candidate *regularity base points* for *ro*'s steps, while uo_5 is excluded because its *commit* phase starts after *ro*'s return point (uo_5 is not excluded in the original *regularity* in [70]). Also, the definition uniquely identifies one *update* operation among those committed before *ro* (uo_2 in our example). uo_1 is excluded because its *commit* phase is not the last one before *ro*'s invocation. Because the commit phases of update executions do not interleave, this candidate is always deterministic.

Finally, we define the meaning of *regular execution* under *SWC* in Theorem 6. Intuitively, the theorem states that a concurrent execution is regular if *i)* every step in every *read-only* operation and in the *traversal* phase of every *update* operation is *valid* (i.e., all operations execute without any "unexpected" behavior); and *ii)* the history of the *update* operations plus one *read-only* operation is *linearizable* (recall that Theorem 5 already proves *linearizability* of the set of the *update* operations).

Theorem 6. *A concurrent execution μ under the SWC model is regular if:*

1. *In the traversal phase of every operation in μ, every step has a base point with some base condition.*
2. *The pre-commit-state of every read-only operation in μ has a regularity base point with some base condition.*

Proof. In the first part of the proof we prove *validity*, which is implied by *1)*.

In fact, assume first an *update* operation u_i in μ. If every step s_i^j in u_i traversal phase has a *base point* with some *base condition*, and by the definition of *base point* and *base condition*, this means that the local state observed by u_i right before executing s_i^j is sequentially reachable. Also, according to Definition 1 and Theorem 5, the local state before the first step in u_i's *commit* phase, s_i^{m+1} (which is *pre-commit-state*$_{u_i}$) also observes a *sequentially reachable* shared state. This means that u_i is safely executing all its steps until s_i^{m+1} without any unexpected behavior. Starting from this step, u_i is executing in isolation because it already started its *exclusive* commit phase, so all the next step until s^n will be also safely executed without unexpected behaviors. Hence, all *update* operations are *valid*.

Now, assume a read-only operation *ro* in μ. In the same sense, by the definition of *base point* and *base condition*, every step in *ro* observes a *sequentially reachable* shared state.

In the second part of the proof we prove that *2)* is sufficient for guaranteeing that the sub-history of $H|\mu$ composed of all write operations in $H|\mu$ enriched with one read-only operation (if any) in $H|\mu$ (which we name $H_{u,r}|\mu$) is linearizable. To do so, we rely on the proof of regularity in [70], given that we define regularity base points in the same way, according to Definition 2: we restrict the candidate *base points* for each read-only operation *ro* to be the *post-state* of either an update operation whose *commit* phase is executed concurrently with *ro* in μ or of the update operation whose *commit* phase is the last one completed before *ro*'s invoke step in μ.

□

In [70], the authors define a *visible mutation point* for an *update* operation as the write step that writes to a shared variable that might be read by a concurrent operation. They prove that a *regular* concurrent execution where each operation has a *single visible mutation point* is linearizable. Proving that the same applies in our model is straightforward (more details are in Section 10.4).

Definition 3. ***(Regular data structures under SWC)*** *A data structure ds is regular if every concurrent execution on ds is regular under the SWC model.*

Theorem 7. *The linked-list in Algorithm 28 is regular.*

Proof. To proof that the linked-list (*ll*) is *regular*, we prove first that every step in every operation in *ll* has a *base condition*. Then we prove that any *concurrent execution* on *ll* complies with the *SWC* model (Theorem 5). And finally, we prove that every *concurrent execution* on *ll* is *regular* (Theorem 6).

The predicates ϕ_i in Algorithm 28 are identified for each step that is reading from the shared memory, in addition to the *return* steps. We first prove that if those predicates are *base conditions*, all the other steps will also have a *base condition* similar to one of those ϕ_i predicates. This is because any other step s^i in any operation O (i.e., *readLast*, *insertLast*, or *removeLast*) performs one of the following actions:

- Writing in the shared memory. By definition, all those steps exist in the *commit* phases of *insertLast* and *removeLast* that are *exclusive* (because the lock is acquired) and *successful* (because the operation cannot perform the first write except in the successful commit). From our assumptions, the first step in those phases has a *base condition* (because it is a read step from the shared memory). Without losing generality, we call this base condition

ϕ_i (specifically, it is ϕ_6 for *insertLast* and ϕ_{11} for *removeLast*). By the definition of *base condition*, every sequential execution starting from a shared state s where $\phi_i(s) = true$ reaches the local state l before the first step in the commit phase. Since the whole execution after this step is *isolated*, then every sequential execution starting from a shared state s where the same $\phi_i(s) = true$ reaches the local state l before any step in the commit phase, and thus ϕ_i is the base condition for all the other writing steps in the *commit* phase.

- Reading from the local state l of the operation. l is the same as the local state after the previous step that reads from the shared memory s^{shread} (or the initial local state $\perp$ if this step does not exist). Thus, in the same way, it has the same *base condition* as s^{shread} (or *true* if the local state is $\perp$).

- Writing on the local state l, which results in a new local state l'. It is clear that if a sequential execution reaches l, then it also reaches l'. So this step can have the same *base condition* as the step before it.

- Acquiring/releasing the locks. This is irrelevant from the shared state of the data structure, and thus it can also use the same *base condition* as the step before it.

Next, we prove that ϕ_i $(i = 1, 2, ..., 11)$ is a base condition for the corresponding step. For the predicates, ϕ_1, ϕ_4, ϕ_7, and ϕ_8, the *base condition* is clearly *true* because they are the first *read* steps from the shared memory in the operations (actually, ϕ_7 may be neglected because the *head* pointer is constant).

For ϕ_9, the local state is as follows: *last* and *secondLast* are constants, and *next* is null. If we start from a shared state where *head.next* is $\perp$, then clearly we will reach line 27 with this local state observed.

For ϕ_2, ϕ_5, and ϕ_{10} (inside the while loop), the local state is as follows: both *last* and *next* are the same, and they are different from $\perp$ (also, in *removeLast*, *secondLast* has the old value of *last*). If we start from a base condition in which *last* is reachable from the head, then any sequential execution will reach the same values of *last* and *next* (and *secondLast* in *removeLast*) at some iteration. This is because the while loops start from the head of the linked list and end when they reach $\perp$.

For (ϕ_3, ϕ_6, and ϕ_{11}), the local state is as follows: *next* is $\perp$, *last* is the last item in the list, and *secondLast* is the one before it. If we start from a shared state such that *last* is reachable from the head, and its next equals $\perp$, then we will reach those steps only when we break the while loops, which means that *next* is $\perp$, and the other two variables are reading the correct values.

Now we prove that any concurrent execution of *ds* is correct under *SWC* by proving that commit phases of update operations are sequential and that for each operation O_i, *pre-commit-state*$_{O_i}$ observes *post-state*$_{O_{i-1}}$. The first claim is trivial, because the commit phases are protected by the global lock *gl*. To prove the second claim, looking at lines 16 and 33 of the *commit* phases of an operation O_i in a concurrent execution μ, the condition at those lines has to be false because it is the *successful commit* phase and not one of the unsuccessful trials, which means that *pre-commit-state*$_{O_i}$ observes $last.next = \perp$. The only way to achieve that is when the *update* operation O_{i-1} committed before O_i either inserts the node observed by *last* in O_i's local state, or deleted the node the node after *last* (making $last.next = \perp$). Thus, O_{i-1} for sure generates a shared state where *last* is reachable from the head, i.e., *pre-commit-state*$_{O_i}$ observes *post-state*$_{O_{i-1}}$ (A special case in *insertLast*, when no item in the list, which is trivial because this way $last = \perp$ and $head.next = \perp$).

The next step is to prove *validity*. We do so by proving that in any concurrent execution μ every step s^i in every operation O has a base point where ϕ_i is true. All the steps except ϕ_9 and those where $\phi_i = true$ requires similar base points (i.e., a post-state of an update operation where the local variable *last* represents a node that is reachable from the head). Intuitively, *last* was linked at some point of the execution to the list (i.e., reachable from the head), or otherwise O would not be able to reach it in the while loop starting from the head. What we want to prove is that this point of execution is one of the *post-states* of the operations. In fact, this point can be the initial state $\mathcal{S}_0$ if the node belongs to the initial state of the list, which becomes the base point of the step (recall that $\mathcal{S}_0$ is accepted as a base point). Otherwise, the point cannot be an intermediate step in any operation simply because *addLast* and *removeLast* have only one write operation. Then, every step whose base condition is ($head \overset{*}{\Rightarrow} last$) has a base point in μ. For the steps where $\phi_i = true$, any *post-state* (including $\mathcal{S}_0$) can be a base point. For ϕ_9, the base point would be either $\mathcal{S}_0$ (if the list is initialized empty) or the *post-state* of the *removeLast* oepration that detached *head.next* from the list (which has to exist or otherwise line 27 would not be reached).

The last step is to prove *regularity*. As we already proved that *ll* is *valid*, the remaining part is to prove that the base point of the *return* step of each *readLast* operation *ro* in any concurrent execution μ is a *regularity base point*. By contradiction, assume that neither the *post-state* of a concurrent *update* operation nor the *post-state* of the operation committed before *ro* is a valid base point of the *return* step. This means that all those states do not observe *last* to be reachable from the head, i.e., *last* was never reachable from the head during the whole execution of *ro*, which contradicts the fact that the while loop starts from the head and ends with *last*.

□

10.2.1 Allowing Concurrent Commits

The implementations of *OSS data structures* usually do not rely on a global lock-based mechanism to finalize the writes, but rather, in order to increase the level of concurrency, the *commit* phase either executes inside TM transactions (hardware or software) [101, 11], or leverages the locking mechanism with *fine-grained* locks that protect (at least) the written locations [48, 54, 59]. Fortunately, some of those techniques provide the same atomicity guarantees as global locks. For example, some TM implementation provides *single lock atomicity (SLA)* guarantees [78] (e.g., the HTM transactions provided by Intel's TSX extensions [65] and the *SLA* version of NORec [25]). By definition, *SLA* guarantees that all the non-transactional reads observe the same serialization of all the concurrent transactions. Thus, if those TM are used to execute the *commit* phases instead of serializing them with a global lock, then we can easily prove that the same guarantees are fulfilled. In fact, in [78] the authors formally prove that executing atomic blocks with *SLA* semantics is equivalent to executing them using *synchronized* blocks protected by a single lock, which implies that our model is safe under this new assumption.

An interesting extension of this direction is to allow TMs with weaker semantics to be used. For example, *Disjoint Lock Atomicity (DLA)* [78] guarantees that only conflicting transactions are observed in the same order. *Asymmetric Lock Atomicity (ALA)* [78] relaxes *DLA* by ordering transactions only by forward dependences. Clearly, data structures that use those semantics cannot be covered by the current version of *SWC* because they break the total order of the *commit* phases, which is required for any concurrent execution to be admitted by *SWC*. At this stage, all those data structures can still be covered by our models if they execute their *commit* phases inside transactions under the *SLA* semantics (e.g., HTM transactions, which is not expected to severely affect the performance because it exploits the efficiency of the HTM hardware components). However, we believe that extending our model to support those semantics is feasible, so we consider that as a future work.

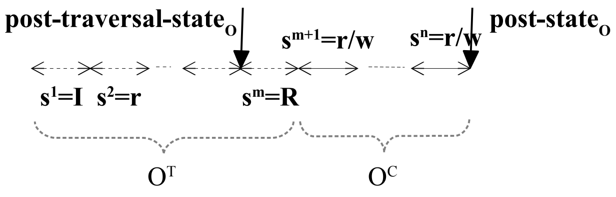
10.3 The Composable Single Writer Commit (C-SWC) Model

We now extend our model by allowing the composition of multiple operations into atomic transactions. For the sake of simplicity, we assume that all the operations belong to the same data structure, then we briefly discuss how this assumption can be relaxed. Algorithm 29 shows an example of a transaction under the *C-SWC* model.

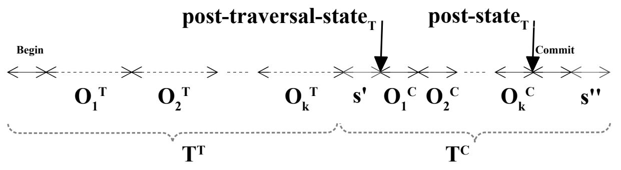
Algorithm 29 An atomic transaction on a composable version of the linked-list of Algorithm 28.

```
1: procedure ATOMIC:$T_1$
2:     $x = 5$
3:     if $readLast() \neq x$ then
4:         $insertLast(x)$
5:     if $readLast() \neq x$ then
6:         ... // illegal execution
7: end procedure
```

In the *C-SWC* model, as shown in Figure 10.3, each operation O_i is split into *traversal* (O_i^T) and *commit* (O_i^C) phases, and the transaction itself is split into a *traversal* phase that combines all the traversal phases of the operations (i.e., $T^T = start \cdot O_1^T \cdot O_2^T \cdot ... \cdot O_k^T$), and a *commit* phase that combines all the commit phases, surrounded by two auxiliary steps to move the execution to/from the exclusive mode (i.e., $T^C = S' \cdot O_1^C \cdot O_2^C \cdot ... \cdot O_k^C \cdot commit \cdot S''$). Like *SWC*, we assume for simplicity that *commit* phases are protected by a single global lock. However, the same arguments adopted in *SWC* can be applied here to consider concurrent executions under the *SLA* semantics. We also assume that the *commit* phases of transactions are the successful ones, and any unsuccessful trial is included in the transaction *traversal* phase.



(a) Operation split



(b) Transaction split

Figure 10.3: Splitting operations and transactions in the C-SWC model.

Figure 10.3 shows how operations are split in the *C-SWC* model. First, the *return* step of each operation is shifted to be the last step of its *traversal* phase. This is important because the return value of the operation may be used later in the transaction (e.g., lines 3 and 5 in Algorithm 29). Second, the auxiliary steps S' and S'' are removed from the *commit* phases of operations and they appear only once in the *commit* phase of the enclosing transaction. Finally, a dummy step $s^{ro-commit}$ is added to the *commit* phase of any *read-only* operation *ro*. This dummy step becomes the only one in the *commit* phase of *ro* because the real *return* step is shifted to the *traversal* phase (as said before).

As shown in Figure 10.3, we define for each operation O_i one more state called *post-traversal-state*$_{O_i}$, which is the local state before O_i's *return* step. We also define for the whole transaction T a state called *pre-commit-state*$_T$ which is the local state after s'.

The first step in formalizing the *C-SWC* model is to refine Definition 1 and Theorem 5 to guarantee that *update* transactions are strict serializable, which also implies that their operations are linearizable.

Definition 4. ***(Executions under C-SWC)*** *In a transactional execution μ with k update transactions whose commit phases are sequential, those k transactions are totally ordered according to the order of their commit phases $\{T_1 \prec_c T_2 \prec_c ... \prec_c T_k\}$. A dummy T_0*

transaction is added such that post-state$_{T_0}$ *is* $\mathcal{S}_0$. *A transactional execution* μ *belongs to the C-SWC model if update transactions have sequential commit phases, and, for every update transaction* T_i *in* μ, *pre-commit-state*$_{T_i}$ *observes the post-state*$_{T_{i-1}}$.

Theorem 8. *Given a transactional execution* μ *with* k *completed update transactions, and the corresponding history of those* k *transactions* $H_k^T|\mu$, *if* μ *is under the C-SWC model, then all the post-states of the* k *transactions are sequentially reachable, and* $H_k^T|\mu$ *is strict serializable.*

Proof. The proof follows the proof of Theorem 5. $H_k^T|\mu$ is *strict serializable* not only *serializable* because the commits in its equivalent sequential history follow the order of the *commit* phases in $H_k^T|\mu$.

□

Corollary 2. *Given a transactional execution* μ *with* k *completed update transactions, and the corresponding history of the operations in those* k *transactions* $H_k|\mu$, *if* μ *is under the C-SWC model, then all the post-states of the operations in those* k *transactions are sequentially reachable, and* $H_k|\mu$ *is linearizable.*

Next, we observe that the definitions of *base points* and *regularity base points* in the *C-SWC* model have to be changed as follows. First, and intuitively, *regularity base points* have to be selected from the *post-states* of the transactions rather than operations, because those are the serialization points in the execution. For *base-points*, it is safe to keep using the *post-state* of the operations like the *SWC* model. This is because any execution should never be subject to any unexpected behavior (e.g., division by zero) if it always observes a shared state that is reachable by a sequential execution (even if this shared state breaks the atomicity of the transactions). Second, any step is seen as a step in a transactional context rather than just an operation.

Definition 5. ***(Base conditions under C-SWC)*** *Given a local state* l *of a transaction* T *on a data structure* ds, *a base condition* ϕ *for* l *is a predicate over the shared state of* ds *where every sequential execution of* T *starting from a shared state* s *such that* $\phi(s) = true$, *reaches* l. *A predicate* ϕ^i *is a base condition for a step* s^i *in an operation* O *if it is the base condition for every local state right before the execution of* s^i *in every sequential execution of every transaction that contains* O.

Definition 6. ***(Base points under C-SWC)*** *An execution of a step* s^i, *with a local state* l, *in a transactional execution* μ *has a base point if there is a sequentially reachable post-state* s, *such that the base condition* $\phi^i(s)$ *holds.*

Definition 7. ***(Regularity base points under C-SWC)*** *A base point* bp *of a step* s^i *in a transaction* T *in an execution* μ *under the C-SWC model is a regularity base point if* bp *is the post-state of either an update transaction whose commit phase is executed concurrently with* T *in* μ *or of the update transaction whose commit phase is the last one completed before the begin step of* T *in* μ *(the initial state is the default).*

Furthermore, unlike the *SWC* model, the consistency of a data structure under *C-SWC* has to include the notion of *internal consistency* of a transaction, which informally means that:

- The *commit* phase of each operation reflects what the operation observed in its *traversal*. For example, in Algorithm 29, if the *readLast* operation in line 3 returns x, the same value x should be the value of the last node in the linked-list when the commit phase of *readLast* is executed during the transaction's *commit*.
- The shared state produced by an operation execution is visible to the subsequent operations in the same transaction. For example, in Algorithm 29 the assertion in line 5 should always pass because, "semantically", lines 3 and 4 guarantee that x is the last element in the list before line 5 (even though the insertion will not be placed into the shared state until the transaction's commit).
- The *return* steps of the operations in the same transaction observe the same shared state.

Those guarantees are formalized as follows:

Definition 8. ***(Internal consistency under C-SWC)*** *A transactional execution μ is internally consistent if for every transaction T_j in μ, the post-traversal-states of every operation O_k have the same base point as pre-commit-state$_{T_j}$.*

Definition 8 intuitively covers all the cases above because it guarantees that there is a shared state from which the execution of any transaction T in μ in isolation will result in the same local states observed before the return of each operation (during the *traversal* phase of T) and before starting the *commit* phase of T. Since the condition in Definition 8 is sufficient (but may not be necessary) for any data structure to be *internally consistent*, it allows the programmer to use the notion of *base conditions* and *base points* to prove the correctness of its design, which is our main goal.

Given the above definitions, the final step is to define *regular* executions under the *C-SWC* model as follows.

Theorem 9. *A transactional execution μ under the C-SWC model is regular if:*

1. *in the traversal phase of every transaction in μ, every step has a base point with some base condition.*
2. *in every transaction T_j in μ, the post-traversal-states of every operation O_k have the same regularity base point as pre-commit-state$_{T_j}$.*

Proof. μ is clearly *internally consistent* because point 2 is the definition of *internal consistency* (Definition 8). The remaining of the proof, which is proving *validity* from point 1 and proving *regularity* from point 2, follows the proof of Theorem 6.

□

The first point adds *validity* similarly to the *SWC* model. The second point in Theorem 9 is the same as Definition 8 except that it restricts the common *base point* to be a *regularity base point*. In fact, our definition of *regularity* is tightly related to the consistency conditions defined for general transactions. Specifically, the second point in Theorem 9 is sufficient for guaranteeing that a sub-history of $H^T|\mu$ composed by all the committed *update* transactions in $H^T|\mu$ plus another transaction in $H^T|\mu$ (i.e., either read-only or update and either live/aborted or committed) is strict serializable. This is similar to *extended update serializability* [2, 87] with the addition of "strictness", i.e., preserving the real-time order. Also, *validity*, that is guaranteed by the first point, is not covered in *serializability*.

Finally, we discuss how operations on different data structures can be executed in the same transaction under the *C-SWC* model as long as the data structures are independent. This is a reasonable assumption because data structures are usually accessed only through their "independent" APIs. Intuitively, if two data structures ds_1 and ds_2 have non-intersecting sets of shared variables X_1 and X_2, and both ds_1 and ds_2 are *regular* under the *C-SWC* model, then we can build a transactional execution that compose their operations without the need to redefine *base conditions* of each (because they are independent). The only conditions in doing that are: *i)* serializing all transactions using the same TM (that guarantees *SLA* semantics); and *ii)* applying the same restrictions on the candidate *base points* and *regularity base points* for each step in the "mixed" transactional executions. Interestingly, if we consider each memory location not included in the *shared state* of any data structure as a special data structure with two operations *Read* and *Write*, then we can define generic transactions with any data structure operations and any memory accesses to follow the *C-SWC* model. That enables the possibility of including techniques like [101, 48] under our model.

10.4 Comparison With Existing Models and Techniques

In this section we overview the intersections between our models and the literature. This comparison helps applying our models to the current designs of concurrent and composable data structures. The first is clearly the relationship with the *SWMR* regularity model [70], which is shown in Theorem 10. Intuitively, since the write operations in a *regular* execution μ under the *SWMR* model are executed sequentially, we can show that μ is also *regular* under the *SWC* model. This is done by considering the *commit* phase of any operation in μ as the operation itself, because this way the definition of *regularity base point* in μ under *SWC* (Definition 2) becomes equivalent to the corresponding definition in *SWMR* [70].

Theorem 10. *For each concurrent execution μ, if μ is regular under SWMR, then μ is regular under SWC.*

Proof. We first identify the split of operations in μ in the *SWC* model, which is straightforward: for each *update* operation *uo*, consider an empty *traversal* phase and a *commit* phase that has all steps of *uo*. The split of read-only operations is identified as usual (only the *return* step is in the *commit* phase). Using this split, μ complies with *SWC* (Definition 1) because it has sequential commit phases, and for each operation O_i, *pre-commit-state*$_{O_i}$ obviously observes *post-state*$_{O_{i-1}}$ (because the whole operations are in the commit phases).

Then we prove that μ is *valid* under *SWC*. In fact, the definition of *validity* is the same in both models, and the observed *base-points* are the same too (because operations have the same sequential order and generate the same *post-state*). Then, as we assume by definition that μ is *valid* under *SWMR*, it is *valid* under *SWC*.

Now we prove that μ is *regular* under *SWC*. Given that μ is *regular* under *SWMR*, every *read-only* operation *ro* observes one of the *post-states* of the operations intersecting with it or the one completed right before it. Consider without losing generality that it is u_i. Since we consider the *commit* phase as the whole operation when we model μ under *SWC*, then the commit phase of u_i is clearly either intersecting with *ro* or the last commit phase completed before it. Hence, u_i is also a *regularity base point* for *ro* under *SWC*. Thus, μ is regular under *SWC*.

$\square$

No previous work discusses *regularity* in *MWMR* data structures. However, *MWMR* regularity is discussed thoroughly for registers in [92]. Our models can be compared to those definitions if we consider that: the shared state of the data structure is a register; each *read-only* operation *ro* is a read from the register that returns *pre-commit-state*$_{ro}$ (which is the local state before *ro*'s *return* step); and each *update* operation *uo* is a write that updates the register to *post-state*$_{uo}$. Briefly, the authors of [92] identified three anomalies that can happen in a *MWMR* concurrent execution. Based on those anomalies, a lattice of *consistency conditions* is built according to the possibility of having each of them. The weakest consistency condition (that allows all the anomalies to happen) is called *MWWeakReg* and is similar to the definition of *regularity* used in the previous sections. The anomalies are briefly described in the following. *1)* For any two reads r_1 and r_2, the set of writes that start before the completion of either r_1 or r_2 is not perceived by both reads as occurring in the same order. A model that disallows this anomaly is said to satisfy the *MWReg* consistency condition. *2)* Two reads performed by the same thread (or process) may be observed in a different order than the one in which they occur at that thread (or process). A model that satisfies *MWReg* and disallows this anomaly is said to satisfy *MWReg+* consistency condition. *3)* Two reads observe different *partial causal order* of all the writes (i.e., a real-time order that takes into consideration the "read-from" relations between reads and writes). A model that satisfies *MWReg+* and disallows this anomaly is proved to be *atomic*.

Theorem 11 proves that executions under our models avoid the first anomaly, and thus satisfy at least *MWReg*. Intuitively, this is because they enforce one permutation of the *update* operations (respectively transactions) to be observed by the non-transactional read steps in the read-only operations (respectively transaction) and the *traversal* phases of the *update* operations (respectively transactions), either by executing the *commit* phases sequentially or by using atomic blocks with SLA semantics.

Theorem 11. *Given a data structure ds as a read/write register, every regular concurrent execution μ on ds under the SWC model satisfies MWReg consistency condition.*

Proof. Since μ is regular under *SWC*, according to Theorem 6, then for each read-only operation ro, the history of all the *update* operations plus ro, $H_{u,r}|\mu$ is linearizable, thus, it is legal and μ-consistent. From the definition of MWMR consistency conditions in [92], this is sufficient for μ to satisfy *MWWeakReg*.

Now we prove that the first anomaly described in Section 10.4 cannot happen in μ, and thus μ satisfies *MWReg*, as follows. Given two *read-only* operations $r1$ and $r2$, since $H_{u,r1}|\mu$ and $H_{u,r2}|\mu$ are linearizable, then they are equivalent to sequential legal histories S_1 and S_2, respectively. Consider without the loss of generality two update operations in μ, u_i and u_j, we prove that the anomaly cannot happen by proving that $u_i \prec_{S_1} u_j$ iff $u_i \prec_{S_2} u_j$. By contradiction, assume without the loss of generality that $u_i \prec_{S_1} u_j$ and $u_j \prec_{S_2} u_i$. Since this is a case of registers, the anomaly can only happen if both u_i and u_j precede both $r1$ and $r2$ in μ. Hence, according to $\prec_{S_1}$, $r1$ observes the value written by u_j. Since the *commit* phases of u_i and u_j follow the SLA semantics (by the definition of SWC), then what $r1$ observes enforces the *commit* of u_i to be completed before the commit of u_j. As u_i and u_j precede $r2$ as well, then $r2$ also observes u_j, which means that $u_i \prec_{S_2} u_j$ (contradiction).

□

Finally, Theorem 12 proves that, similar to the $SWMR$ model, adding $SVMP$ is sufficient for a data structure to be atomic.

Theorem 12. *In both SWC and C-SWC models, if each update operation (respectively transaction) in each concurrent (respectively transactional) execution has a single visible mutation point, then the data structures under those models are atomic.*

Proof. The theorem can be proven in a similar way to the corresponding one in the $SWMR$ model. We prove that for the concurrent executions, as transactional executions can be similarly proven. By definition, a *single visible mutation point* means that the *commit* phase of every *update* operation has at most one *write* step that affects the return values of the concurrent *read-only* operations. Let us assume the history $H|\mu$ of all the operations in a concurrent execution μ. We prove the thesis by proving that if μ is *regular* and each operation in μ has a *SVMP*, then $H|\mu$ is linearizable. First, as μ is *regular*, then the history of the update operations $H_u|\mu$ is linearizable. This means that there is a legal sequential history S that is equivalent to $H_u|\mu$ (i.e., preserves the real-time order of the updates in μ). If we extend S to S' by adding all the *read-only* operations as follows: if the last $SVMP$ that appears before the *return* step of a *read-only* operation ro belongs to an *update* operation u_i, then ro is ordered in S' after u_i. Assume that each read-only operation in S' returns the same value as $H|\mu$, we will prove that S' is legal and equivalent to $H|\mu$ (which implies that $H|\mu$ is linearizable). To prove that, since the return step of ro in μ observes the shared state after the $SVMP$ of u_i, and ro comes after u_i in S', then the return values in both S' and $H|\mu$ are the same, which is a valid *post-state* in μ. Thus S' is legal. The order of the *read-only* operations in S respects the real-time order in $H|\mu$ because of the following. First, two *update* operations trivially respect the real-time order because they admit the order of the *commit* phases in μ. For two reads, if their return values come after the same *SVMP*, then they can be ordered in any way in S', so we can select the order that respects their real-time order in $H|\mu$. If they are not after the same *SVMP*, then the update operation that includes this $SVMP$ will enforce their real-time order in S'. Finally, for a read-only ro operation with an *update* operation u_i, assume that the update operation whose $SVMP$ is observed by ro is u_j. If $u_i = u_j$, then u_i and ro are intersecting and can be ordered any way in S'. Otherwise, ro will come after u_j in S', which obviously puts ro in the correct real-time order with respect to u_i (because if u_i is after (respectively before) u_j it can be safely after (respectively before) ro in S'.

$\square$

Similar to our models, the *LS-Linearizability* [41] model adds the notion of *validity* to *linearizability* by defining the term *local serializability.* It roughly means that each process observes a local serialization at the level of the operation steps. Although *local serializability* achieves the same goal as *validity*, it is more conservative because it requires that all the steps of an operation belong to the same *sequential execution.*

On the implementation side, we claim that many concurrent and composable data structures, beyond OTB, can be covered by our models. Here we report an example of each:

- **Lazy linked-list** [54]. This is the closest implementation to the example we give in Algorithm 28. In fact, it extends Algorithm 28 by allowing the queries, insertions, deletions to be at any place of a *sorted* list. This is mainly achieved by tracking two nodes (*pred* and *curr*) instead of the *last* node we track in Algorithm 28. Interestingly, lazy linked-list uses the same split of the operations and a similar validation mechanism at the beginning of the *commit* phase, which allows a straightforward extension. The original design of the lazy linked-list uses fine-grained locks at commit time. For using the courrent models, this can be easily replaced with HTM transactions, as we discussed before. In future, if we allow weaker TMs with *ALA* [78] semantics, the exact original design can be modeled with *SWC.*

- **Partitioned Transactions (ParT)** [101]. This is a methodology that composes data structures operations using the exact split mechanism we propose in *C-SWC.* It calls the *traversal* phase as *planning* phase and the *commit* phase as *completion* phase. Interestingly, ParT proposes using HTM transactions (among other alternatives) for executing the *completion* phases, so it complies with our assumption that *commit* phases are executed using an *SLA* TM.

OTB is similar to ParT except that it uses fine-grained locks to guard *commit* phases. As the lazy linked-list, OTB fits with our models by replacing those locks with HTM transactions.

10.5 Summary

We addressed the problem of providing a unified model for the current designs of concurrent and composable data structures. Specifically, we targeted the set of *OSS data structures* and we presented two models: *SWC*, which can be used to model and prove the correctness of the concurrent data structures on this set; and *C-SWC*, which extends *SWC* to cover the composable designs.

Chapter 11

Conclusions

In this dissertation we proposed contributions aimed at identifying and addressing the challenges of designing and modeling transactional data structures.

The first challenge to be addressed is to allow an atomic execution of multiple data structure operations, named *composability*. To address this challenge we presented Optimistic Transactional Boosting (OTB), a novel methodology for boosting concurrent data structures to be transactional. We deployed OTB on a number of concurrent data structures, by producing the following transactional versions:

- Linked-list-based and Skip-list-based set;
- Skip-list-based priority queue;
- TxCF-Tree, an efficient transactional balanced tree that, in addition to the OTB advantages, it also provides a minimal interference between its structural operations (i.e., rotations and physical removals) and its semantic operations (i.e., logical insertions, deletions, and queries).

OTB-based data structures showed excellent results when compared to the original boosting methodology. In fact, they perform close to the optimized concurrent (non-transactional) version of the data structure. They also perform up to an order of magnitude better than pure-STM-based data structures implementations.

The second challenge is to integrate transactional data structures with both STM and HTM transactions. To address this challenge, we integrated OTB-based data structures with both DEUCE and RSTM framework, while enabling HTM transactions in the latter. Our integration allows executing both memory-level and semantic-level reads/writes in the same transaction without losing the performance gains guaranteed by OTB. Experimental results show that the performance of the modified frameworks is improved by up to 10x over the original frameworks.

In the same line of addressing the *integration* challenge, we aim at enhancing the overall STM performance since OTB becomes part of generic TM frameworks. We do so by presenting Remote Transaction Commit (RTC) and Remote Invalidation (RInval), two new algorithms which use an efficient remote core locking mechanism and optimize transactions' critical path. RTC outperforms other STM algorithms, by up to 4× in high contention workloads. This is mainly because executing commit phases in dedicated server cores alleviates the overhead of cache misses, CAS operations, and OS descheduling. We also showed that executing two independent commit routines in two different server cores results in up to 30% improvement.

RInval applied the same idea of RTC on invalidation-based STM algorithms. Additionally, to optimize the validation/commit overheads, RInval splits commit and invalidation routines and runs them in parallel on different servers. As a result, RInval performs up to 2× faster than the corresponding validation-based STM algorithm (NOrec) and up to an order of magnitude faster than the corresponding invalidation-based STM algorithm (i.e., InvalSTM).

The last challenge addressed in this dissertation is the definition of a unified model for assessing the correctness of the OTB-based data structures, which has been fulfilled by introducing two models: one regarding concurrent data structures (named SWC), and one related to transactional data structures (named C-SWC). SWC and C-SWC can be exploited to model a wider set of data structure than the former models in literature. We justified that by showing in detail how to model a linked-list-based set using SWC. Then, we discussed how to apply them on the more complex (OTB-Based) data structures.

11.1 Summary of Contributions

To summarize, this dissertation made the following contributions:

- OTB, an optimistic methodology to efficiently allow a transactional access to the highly concurrent data structures.
- A complete design and implementation of two transactional list-based data structures using the OTB methodology, namely set and priority queue.
- TxCF-Tree, an efficient transactional balanced tree with an optimized communication between its structural and semantic operations.
- The extended versions of two TM frameworks (DEUCE and RSTM) to support the integration of OTB data structures with generic *memory-based* TM transactions. The extension of RSTM also allows the integration with HTM transactions.
- RTC and RInval, the first STM algorithms that exploit more advanced locking mechanisms than spin locking.

- SWC and C-SWC, two models to infer the correctness of the concurrent and transactional data structures in a unified way.

11.2 Future Work

As a future work we suggest designing more data structures using OTB methodology. Having a library of OTB-Based data structures, similar to `java.concurrent.util` library, allows legacy applications to take advantage of the optimized designs of those data structures, especially if the integration with TM frameworks is also encapsulated in such libraries.

RTC (and RInval) can be used as a fall-back to Haswell's HTM transactions. RTC centralizes the commit phases inside its servers, which allows minimum interference with the hardware fast-path. Additionally, and more importantly, the same improvements we achieved in Chapter 7 by injecting OTB semantics in the slow-path can be achieved when RTC is used in that slow-path. This way, we further improve the performance by merging the benefits of OTB and RTC. Finally, RTC servers can be efficiently used for further improvements, like profiling the committed/aborted transactions and analyzing the abort messages (being either conflict, capacity, external aborts, ...). This analysis can be used as a guideline for further improvements (e.g., batching STM transactions).

SWC and C-SWC can be used to model more complex data structures than the example shown in the dissertation, such as TxCF-Tree. The models themselves can also be extended to support multiple writers' commit. Finally, the models can be extended to cover generic transactions not only data structure operations.

Bibliography

[1] Rochester software transactional memory runtime. www.cs.rochester.edu/research/synchronization/rstm/. URL `www.cs.rochester.edu/research/synchronization/rstm/`.

[2] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[3] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, 2011.

[4] Y. Afek, A. Levy, and A. Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 295–296. ACM, 2013.

[5] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed Computing*, 27(6):393–417, 2014.

[6] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: an automated transactional approach to concurrent memory reclamation. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, page 25, 2014.

[7] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11.*, pages 316–327. IEEE, 2005.

[8] M. Arbel and H. Attiya. Concurrent updates with RCU: search tree as an example. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 196–205, 2014.

[9] H. Avni and B. Kuszmaul. Improve htm scaling with consistency-oblivious programming. *TRANSACT 14: 9th Workshop on Transactional Computing.*, March, 2014.

[10] H. Avni and A. Suissa. Tm-pure in gcc compiler allows consistency oblivious composition. *WTTM 13: 5th Workshop on the Theory of Transactional Memory*, October, 2013.

[11] H. Avni and A. Suissa-Peleg. Brief announcement: Cop composition using transaction suspension in the compiler. In *Proceedings of the 28th International Conference on Distributed Computing*, DISC '14, page 550552, 2014.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, 2010.

[14] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 6–15. ACM, 2010.

[15] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 329–342, 2014.

[16] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *ISCA*, pages 225–236, 2013.

[17] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, 2014.

[18] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *9th Workshop on Transactional Computing*, TRANSACT '14, 2014.

[19] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. *TRANSACT 14: 9th Workshop on Transactional Computing.*, March, 2014.

[20] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 56–67. ACM, 2007.

[21] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 161–170. ACM, 2012.

[22] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 229–240, 2013.

[23] L. Dalessandro and M. L. Scott. Sandboxing transactional memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 171–180. ACM, 2012.

[24] L. Dalessandro, D. Dice, M. L. Scott, N. Shavit, and M. F. Spear. Transactional mutex locks. In *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, pages 2–13, 2010.

[25] L. Dalessandro, M. Spear, and M. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78. ACM, 2010.

[26] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 39–52, 2011.

[27] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. In *ASPLOS*, pages 39–52, 2011.

[28] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[29] M. David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004.

[30] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[31] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.

[32] D. Dice, T. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *6th Workshop on the Theory of Transactional Memory*, WTTM '14, 2014.

[33] N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing*, ICAC '14. USENIX Association, 2014.

[34] D. Drachsler, M. T. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 343–356, 2014.

[35] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 131–140, 2010.

[36] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 257–266, 2012.

[37] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[38] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd International Symposium on Distributed Computing (DISC)*, pages 93–107. Springer, 2009.

[39] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[40] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 101–110. ACM, 2010.

[41] V. Gramoli, P. Kuznetsov, and S. Ravi. Brief announcement: From sequential to concurrent: correctness and relative efficiency. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 241–242, 2012.

[42] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[43] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 404–415, 2009.

[44] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. K. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS*, pages 1–13, 2004.

[45] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[46] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, pages 300–314, 2001.

[47] A. Hassan, R. Palmieri, and B. Ravindran. On developing optimistic transactional lazy set. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, pages 437–452, 2014.

[48] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP), Poster paper*, 2014.

[49] A. Hassan, R. Palmieri, and B. Ravindran. Integrating transactionally boosted data structures with stm frameworks: A case study on set. In *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[50] A. Hassan, R. Palmieri, and B. Ravindran. Remote invalidation: Optimizing the critical path of memory transactions. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[51] A. Hassan, R. Palmieri, and B. Ravindran. Transactional interference-less balanced tree. In *Proceedings of the 29th International Symposium on Distributed Computing*, DISC '15, 2015.

[52] A. Hassan, R. Palmieri, and B. Ravindran. Remote transaction commit: Centralizing software transactional memory commits. In *IEE Transactions on Computers, To appear*, 2015.

[53] A. Hassan, S. Peluso, R. Palmieri, and B. Ravindran. On the correctness of optimistic composable data structures. In *WTTM 15: 7th Workshop on the Theory of Transactional Memory*, 2015.

[54] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. *Principles of Distributed Systems*, pages 3–16, 2006.

[55] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[56] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1): 124–149, 1991.

[57] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[58] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[59] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012.

[60] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[61] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 522–529, 2003.

[62] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[63] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, pages 161–171, 2012.

[64] Intel Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[65] J. Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 2013.

[66] G. Kestor, R. Gioiosa, T. Harris, O. Unsal, A. Cristal, I. Hur, and M. Valero. Stm2: A parallel stm for high performance simultaneous multithreading systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011*, pages 221–231. IEEE, 2011.

[67] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[68] L. Lamport. On interprocess communication. part II: algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[69] K. S. Larsen. Avl trees with relaxed balance. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 888–893. IEEE, 1994.

[70] K. Lev-Ari, G. Chockler, and I. Keidar. On correctness of data structures under reads-write concurrency. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 273–287, 2014.

[71] J. Lozi, F. David, G. Thomas, J. L. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 65–76, 2012.

[72] V. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236. ACM, 2008.

[73] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[74] A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 59–71.

[75] A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

[76] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[77] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 314–325. ACM, 2008.

[78] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2008.

[79] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[80] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[81] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization, 2008. IISWC 2008.*, pages 35–46. IEEE, 2008.

[82] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Austin: IEEE Computer Society, 2006.

[83] J. Moss and A. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[84] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 317–328, 2014.

[85] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.

[86] M. Olszewski, J. Cutler, and J. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *16th International Conference on Parallel Architecture and Compilation Techniques, 2007. PACT 2007.*, pages 365–375. IEEE, 2007.

[87] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Brief announcement: Breaching the wall of impossibility results on disjoint-access parallel tm. In *Proceedings of the 28th International Symposium on Distributed Computing*, DISC '14, pages 548–549, 2014.

[88] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 53–64.

[89] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 284–298, 2006.

[90] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 221–228. ACM, 2007.

[91] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *SPAA*, pages 53–64, 2011.

[92] C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.*, 40(1):28–62, 2011.

[93] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[94] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10 (2):99–116, 1997.

[95] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris. Performance analysis of concurrent red-black trees on htm platforms. *TRANSACT 15: 10th Workshop on Transactional Computing.*, June, 2015.

[96] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 273–283. ACM, 2010.

[97] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 338–339, 2007.

[98] M. F. Spear, M. M. Michael, and C. von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 275–284. ACM, 2008.

[99] TM Specication Drafting Group. Draft specification of transactional language constructs for c++, version 1.1, 2012.

[100] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[101] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 76–86, 2015.