On Improving Transactional Memory: Optimistic Transactional Boosting, Remote Execution, and Hybrid Transactions

Ahmed Hassan

Preliminary Examination Proposal 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

Binoy Ravindran, Chair
Chao Wang
Jules White
Robert P. Broadwater
Eli Tilevich

May 8, 2014
Blacksburg, Virginia

Keywords: Transactional Memory, Transactional Boosting, Remote Execution, Hybrid Transactions, Semantic Validation

On Improving Transactional Memory: Optimistic Transactional Boosting, Remote Execution, and Hybrid Transactions

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 synchronization from the programmer to an underlying framework using hardware (HTM) and/or software (STM) components.

Although STM provides a generic solution for developing more complex concurrent applications, it suffers from scalability challenges due to the intensive speculation and meta-data handling. Moreover, In specific components such as transactional data structures, this intensive speculation results in more false conflicts than the optimized concurrent data structures.

On the other hand, as computer architects and vendors chose the design of simple best-effort HTM components, HTM imposes limitations on the transactions and provides no guarantees for the transactions to eventually commit in hardware. These limitations raise the need to move into two (orthogonal) directions. The first direction is to design efficient hybrid TM frameworks which minimize the overhead of any HTM/STM interconnection. The second direction is to enhance the performance of the fall-back STM paths themselves, including enhancing specific components such as transactional data structures.

In this dissertation, we propose two approaches to enhance the overall TM performance. First, we design an optimistic methodology for transactional boosting to specifically enhance the performance of the transactional data structures. Second, we enhance the performance of STM in general by dedicating hardware cores for executing specific parts of the transactions like commit and invalidation. In both approaches, we first show a pure software solution. Then we propose a hybrid TM solution which exploits the new HTM features of the currently released Intel's Haswell processor.

Optimistic transactional boosting (OTB) is a methodology to design a transactional versions of the highly concurrent lazy data structures. An earlier (pessimistic) boosting proposal adds a layer of abstract locks on top of the optimized concurrent data structures. We propose an optimistic methodology of boosting which allows greater data structure-specific optimizations, easier integration with STM frameworks, and lower restrictions on the boosted operations than the original pessimistic boosting methodology.

Based on our proposed OTB methodology, we implement transactional versions of both set and priority queue. We also extend the design of DEUCE, a Java STM framework, to support OTB integration. 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.

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 major proposed post-preliminary research is to exploit HTM, using the *transactional synchronization extensions* of Intel's Haswell processor (TSX), to enhance both OTB and RTC. OTB can be enhanced by executing its commit phase in hardware transactions. This approach shows significant improvements in simialr work in literature like Consistency Oblivious Programming (COP), and Reduced Hardware Transactions (RH). RTC can be used as an efficient *centralized* fall-back path to Haswell's HTM transactions. Centralizing the fall-back path in RTC's server allows different enahncements, such as profiling the overall HTM retries and estimating the best fall-back alternatives. Additional directions include designing more OTB data structures, such as maps and balanced trees. Another direction is to involve the contention manager in the decisions made by the servers in both RTC and RInval.

This work is supported in part by US National Science Foundation under grant CNS 1116190, and by VT-MENA program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the beginning of the new century, computer manufacturers faced difficulties in increasing CPU's clock speed because of hitting the overheating wall. As a result, multi-core architectures has been introduced as an alternative for exploiting the increasing number of transistors that can fit into the same space (according to Moore's low). In multi-core architectures, increasing the performance is achieved by adding more cores rather than increasing CPU's frequencies.

Since this new turn to multi-core machines, software developers were enforced to develop *concurrent programs* in which they have to synchronize the pieces of code that access the same shared memory simultaneously (i.e. *critical sections*). Programmers used to protect their critical sections using locks. On the hardware level, implementing efficient locks motivates adding more complicated *atomic* instructions such as *compare and set* (CAS) operations. However, on the software level, synchronization using locking is not an easy task. On the one hand, coarse-grained locking, in which the shared objects are synchronized using a single global lock, is easy to program but it minimizes concurrency and does not utilize the multi-core architectures. 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 problems 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 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.

Additionally, fine-grained locking raises another serious problem called *composability* –i.e. composing two efficient pieces of (concurrent) code in one atomic block. For example, in highly concurrent data structures, like lazy and lock-free data structures [39], allowing two operations to execute atomically (e.g. two add operations on the same linked-list, or an addition to a linked-list and a removal from another linked-list) requires non-trivial modifications

in the data structure code.

1.1 Transactional Memory

Transactional memory [38] (TM) is 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 [66, 19, 65, 23, 59, 24, 47, 57, 58], pure hardware [38, 5, 29, 54], and hybrid [21, 60, 20] 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 overheads and allows the programmer to write large and complex concurrent applications with high correctness and progress guarantees.

Transactional memory is increasingly gaining traction: Intel has released a C++ compiler with STM support [41]; Oracle [15, 68], AMD [2, 16], IBM [12], and Intel [42] have released experimental or commodity hardware with transactional memory support; GCC has released language extensions to support STM [67].

Having common and standard API's 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 the 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 [66] and NOrec [19], compacted using bloom filters [9] as in RingSTM [65], or fine-grained using ownership records as in TL2 [23] and TinySTM [59]. 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 also have programmability challenges. They vary in properties such as progress guarantees, publication and privatization safety [52, 64], support for nesting [55, 69], 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.

Motivated by these observations, 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. In this dissertation, we study these overheads, analyze the effect of each one on transactions' execution time, and propose novel solutions to reduce them.

The first overhead is meta-data handling. An STM transaction can be viewed 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. 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 *remote invalidation* (RInval), a novel STM algorithm, which alleviates the

overhead on the critical path.

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 [44] and RSTM in C/C++ [48, 1], use spin locks in their STM algorithms to protect memory reads and writes. The evaluation of some recent locking mechanisms, like flat combining [36] (FC) and remote core locking [46] (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). We present a novel STM algorithms called *remote transaction commit* (RTC) which is, to the best of our knowledge, the first STM algorithm that make use of these enhanced locking mechanisms. RTC's principles have been subsequently integrated in RInval as well.

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. Transactional memory is prone to false conflicts by nature 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 clear in the context of transactional data structures, which we cover with more details in Section 1.2.

1.1.2 Hardware Transactional Memory

TM was firstly proposed in hardware rather than in software, as a modification in the cache coherence protocols [38]. 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 multi-core architectures. The lack of commercial CPU's with HTM support enforced the research studies to rely on simulations [29] to evaluate to proposed HTM architectures.

Recently, an important milestone in TM has been achieved by supporting HTM in both IBM's and Intel's processors [12, 42]. The common design principle in these releases (as well as AMD's proposal which will be released soon [16]) is the principle of *best-effort execution*. There is no guarantee that HTM transactions will eventually commit, whatever 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 reason of failure is what we call *capacity failure*, which means that transaction's footprint (basically its reads and writes) does not fit in L1 cache which is private to each physical core and is used by TSX to buffer transactional reads and writes. Other reasons include page faults, system calls invocations, ..., etc.

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, 13] or STM mechanisms [60, 20, 49]. As another direction, HTM has been proposed to specifically enhance the design of concurrent and transactional data structures [6, 7]. In our post-preliminary work, we propose exploiting Intel's TSX extensions to enhance our approaches.

1.2 Transactional Data Structures

The increasing ubiquity of multi-core processors motivates 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*. 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) [62] 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 1.1 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 [39], do not suffer from this false conflict.

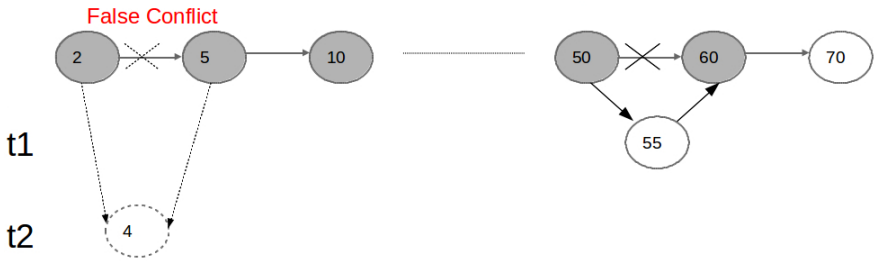


Figure 1.1: An example of false conflicts in a linked-list.

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 [25], open nesting [56], and early release [40]. Another direction is to use STM frameworks as is but more efficiently to design libraries of data structures. Examples in this direction include transactional collection classes [14], transactional predication [11], and speculation friendly red-black tree [17].

Another appealing alternative is Herlihy and Koskinen's *Transactional Boosting* [37] methodology, which converts concurrent data structures to transactional ones by providing a *semantic layer* on top of existing concurrent objects. The semantic layer deals with memory locations according to the semantic meaning of the data structures, unlike the native memory layer which synchronizes memory accesses independent from their semantic meaning. This layer uses abstract locks on the items managed by the transactional operation and populates the so called *semantic undo-log* with inverse operations in order to annul the effect of already performed actions in case of abort. Abstract locking affects only the accessed items instead of the whole data structure, according to the semantics of the data structure itself. Such fine-grained (abstract) locking potentially enables executing commutative operations in parallel.

Although Herlihy and Koskinen's boosting technique allows transactional operations, it has downsides that limit its applicability. First, abstract lock acquisition and modifications in memory are eager. This eager writing contradicts with the methodology of most STM algorithms, which makes the integration between the "boosted" data structures and the lazy STM frameworks difficult. Second, the technique uses the underlying concurrent data structure as a black box, which prevents optimizations on the new transactional characteristics of the objects. Finally, the methodology requires defining an inverse for each operation, which is not natively supported in all data structures. To overcome these downsides, we present an optimistic methodology for boosting concurrent collections, called *optimistic transactional boosting* (OTB). OTB allows greater data structure-specific optimizations, easier integration with STM frameworks, and lower restrictions on the boosted operations than the original boosting methodology.

1.3 Summary of Current Research Contributions

According to our analysis of the overheads in current STM approaches, we followed two main directions to enhance STM's performance. The first direction is to reduce the false conflicts by implementing efficient transactional data structures based on the idea of transactional boosting, and integrating these data structures in STM frameworks. The second direction is to optimize the critical path of the transactions execution.

In the first direction, we design *Optimistic Transactional Boosting* (OTB) [32], an optimistic methodology for converting concurrent data structures to 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, optimistic boosting combines the benefits of concurrent data structures (i.e., un-monitored traversals), boosting (i.e., semantic validation), and transactional memory (i.e., optimistic conflict detection).

OTB gains significant advantages over Herlihy and Koskinen's boosting, which we call "pessimistic" boosting hereafter due to its pessimistic behavior on lock acquisition. First, it avoids the need for defining inverse operations. Second, it uses the concepts of validation, commit, and abort in the same way as general (optimistic) STM algorithms, but at the semantic layer, which enables easy integration with STM frameworks. Finally, it uses highly concurrent collections as white boxes (rather than black boxes as in pessimistic boosting) for designing new transactional versions of each concurrent (non-transactional) data structure, which allows more data structure-specific optimizations.

Next, we show how to integrate transactionally boosted data structures with the current STM frameworks [33]. More specifically, we show how to implement OTB data structures in a standard way that can integrate with STM frameworks, and how to modify DEUCE STM framework [44] 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 the 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.

To evaluate OTB, we first show that optimistic boosted objects outperform pessimistic boosted objects in most of the cases. Then we show that integrating OTB set into DEUCE framework does not affect this performance gain. Our evaluation shows that performance is

improved by up to an order of magnitude when we use OTB-Set instead of a pure STM set inside DEUCE framework.

In the second direction, we present *Remote Transaction Commit* (RTC), 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 [22], hardware overheads such as CAS operations and cache misses are critical bottlenecks to scalability in multi-core 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).

To evaluate RTC, we analyze STAMP applications [53] to show the relationship between commit time ratio and RTC's performance gain. Then, through experimentation, we show that RTC has low overhead, peak performance for long running transactions, and significantly improved performance for high number of threads (up to 4x better than the other STM algorithms). We also illustrate RTC's correctness and the impact of increasing the number of RTC servers on performance.

Next, we present *Remote Invalidation* (RInval) [34], an STM algorithm which applies the same idea of RTC in invalidation-based STM algorithms [27]. 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.

To evaluate RInval, we analyze the parameters that affect the critical path of STM transaction execution, and study the effect of using either validation or invalidation mechanism on the overall performance of both micro-benchmarks and the STAMP benchmark. Then, we show how RInval alleviates these overheads on the critical path. Through experimentation, we show that RInval outperforms past validation-based (NOrec) and invalidation-based (InvalSTM) algorithms, on both micro-benchmarks and the STAMP benchmark [53], yielding performance improvement up to 2x faster than NOrec and an order of magnitude faster than InvalSTM.

1.4 Proposed Post Preliminary-Exam Work

After the Preliminary Examination, we propose to seek further enhancement of OTB, RTC, and RInval by exploiting Intel's TSX extensions.

OTB operations cannot be put as a whole inside HTM atomic blocks, because it has an unmonitored traversal phase which should be kept away from any STM and/or HTM speculation. However, OTB can be significantly enhanced if the monitored commit part is executed inside HTM blocks instead of being executed using software lock-based mechanisms.

RTC (and Rinval) has an important advantage of centralizing the commit phases inside the servers. This allows better communication between HTM transactions and RTC servers if we use RTC as a fall-back path to the HTM transactions. Furthermore, RTC servers can be used to batch the software transactions to reduce their interconnection with the fast hardware transactions. The servers can also be used to profile transactions' aborts and provide the whole system with the optimal fall-back mechanism.

An additional post-preliminary direction is to design more OTB data structures, such as maps and balanced trees. Our goal is to show how OTB methodology is general enough to be applied on different types of data structures, and to design a transactional library of the most common data structures.

We also plan to enhance the performance of RTC and RInval by involving the contention manager in the decisions of the servers. Our evaluation shows that the performance in some cases (like long linked-lists) is significantly affected by the contention manager and its decisions.

As our final goal, we want to include all our enhancements (OTB, RTC, and RInval, with both STM and HTM versions) in a unique TM framework. For achieving that, our targets are RSTM C/C++ framework, and DEUCE Java framework.

1.5 Proposal Outline

The rest of the proposal is organized as follows. We overview past and related work in Chapter 2. We describe OTB methodology and how we applied it on different data structures in Chapter 3. Then, in Chapter 4, we show how we integrate OTB with DEUCE framework. Chapters 5 and 6 describe our RTC and RINval STM algorithms, respectively. Chapter 7 concludes the thesis and further discusses proposed post-preliminary work.

Chapter 2

Past and Related Work

2.1 STM algorithms

Since the first proposal of software transactional memory [62], many STM algorithms with different design decisions were introduced in literature. In this section, we overview past STM algorithms that are most similar or relevant to RTC and RInval, and contrast them with our algorithms. Basically, RTC is a validation-based algorithm which extends NOrec [19], and RInval is an invalidation-based algorithm which extends InvalSTM [27]. 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 [19] 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 *timestamp* is odd, this means that some transaction is executing its commit. 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 means *no ownership records*). Orecs were used in many STM

algorithms [23, 59] as the meta-data associated with each memory block to detect and resolve conflicts between transactions. Some other STM algorithms, such as RingSTM [65], 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.

We will show in Chapter 5 that RTC inherits all the strong properties of NOrec (e.g., reduced false conflicts, minimal locking overhead, and easy integration with hardware transactions), and also solves the problem of executing independent commit phases serially, by adding dependency detector servers to run the commit routines which are independent from the main server's commit routine.

2.1.2 InvalSTM

The invalidation approach has been presented and investigated in earlier works [30, 40, 27]. Among these approaches, Gottschlich *et. al* proposed commit-time invalidation, (or InvalSTM) [27], an invalidation algorithm that completely replaces version-based validation without violating opacity [28].

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 [9] 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 [19]. 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 [28].

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).

RInval solves the problem of InvalSTM by running commit and invalidation routines separately in different server cores. Additionally, it uses an efficient locking mechanism similar to RTC. In RInval, conflicts will always be solved by aborting the conflicting transactions rather than the committing transaction. Although this may result in reducing the efficiency of the contention manager, it opens the door for more significant improvements, such as parallelizing commit with invalidation, as we will show later in Chapter 6.

2.1.3 Other STM algorithms

RingSTM [65] introduced the idea of detecting conflicts using bloom filters [9]. 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. In RTC, as we will show in details later, bloom filters are only used to detect dependency between transactions, while validation remains value-based like NOrec. Also, scanning the bloom filter is now the responsibility of servers, and does not waste the time of application

threads.

TL2 [23] 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.

InvalSTM is not the only work that targets reducing the cost of incremental validation. DSTM [40] is an example of partial invalidation which eagerly detects and resolves write-write conflicts. STM^2 [43] 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 [18].

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 [51]). 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 [48, 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 Remote Core Locking

Some work that is similar to RTC (and RInval) exists in the literature. But none of them has been specifically tailored for STM systems. In particular, Remote Core Locking (RCL) [46] 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 [26].

RCL outperforms traditional locking algorithms like MCS [51] and Flat Combining [36] 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 [36], 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.

2.3 Transactional Boosting

Herlihy and Koskinen's transactional boosting methodology [37] 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, for which our OTB methodology proposes solutions as shown in details in Chapter 3:

- Transactions with eager writes do not natively guarantee Opacity [28]. Before a transaction finishes and commits, other transactions can see a partial state of the objects. In addition, although aborted transactions will roll back their operations, eager writes can be read by any other transaction before aborts take place. The effect of such doomed transactions [61] can lead to unexpected behaviors such as infinite loops and illegal memory accesses. Some STM algorithms deal with doomed transactions using mechanisms such as sandboxing [18]. Although abstract locking in pessimistic boosting 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.
- The pessimistic boosting approach 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.
- Pessimistic boosting 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.

Semantic conflict detection has been previously studied in different ways other than transactional boosting. The technique of open nesting [56] is used in some STM frameworks as a way to isolate data structure operations in nested transactions. Open-nested transactions commit and abort independently from their parent transactions. However, open nesting, like pessimistic boosting, releases writes early on shared memory, which breaks isolation on other levels of nested transactions. Additionally, open nesting still uses the STM framework for synchronization, which means that they incur the overhead of unnecessary monitoring. Early release [40] and elastic transactions [25] attempt to minimize this overhead by giving programmers control on removing items from read-sets. Unlike optimistic boosting, these

techniques, however, work at the memory level and increase the burden on programmers, which conflicts with the simple user interface of transactional memory. Abstract nested transactions (ANT) [31] allow object operations inside nested transactions to re-run independently from the enclosing transaction, which reduces the effect of false conflicts. However, ANT does not provide any optimizations for the data structure operations themselves.

Transactional predication [11] uses the underlying STM framework only in semantically conflicting phases, and bypasses STM in the traversal phases, which are the main source of false conflicts. However, the technique still uses the STM framework as a black box, and its predicate abstraction is only suitable for data structures whose operations are commutative on the keys, unlike priority queue, for example. Transactional Collection Classes (TCC) [14] use nested transactions (both open- and closed-nested) to wrap concurrent collection classes in long running transactions. However, TCC inherits the performance overheads of both transactional memory and nesting. Crain *et. al* proposed a speculative-friendly red-black tree [17] with the idea of decoupling rotation parts in separate transactions, to reduce the overhead of false conflicts. However, it still uses STM and inherits its overhead.

Chapter 3

Optimistic Transactional Boosting

3.1 Methodology

Optimistic transactional boosting (OTB) [32] is a methodology to boost lazy data structures to be transactional. The common feature of 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 usually followed by a validation step, and then a 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 two main guidelines.

- *Rule 1: Each data structure operation is divided into three steps.*
  Traversal. This step scans the objects, 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 in this step. The step is repeated after each new read (to guarantee opacity [28]) and at commit time. **Commit.** This step performs the modifications to the shared data structure. The step is not done at the end of each operation, but is deferred to 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 write-sets, classical (semantic) two-phase locking is used. This semantic (or abstract) locking prevents semantic conflicts at commit.
- *Rule 2: Data structure design is adapted to support transactions.*
  This rule has two main objectives. First, part of the correctness of transactional data structures depends on the sequence of operations executed in the same transaction. Data structure design has to be adapted to guarantee this part, which is not natively

guaranteed by the basic lazy data structure design. For example, if an item is inserted in a set twice inside the same transaction, the second insertion has to return false (indicating that the item is in the set, as a result of the first operation), although the item is not yet physically inserted in the set (because all physical modifications are deferred to transaction commit). Second, additional optimizations can be achieved because the data structure is now transactional. For example, 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 read/write sets in STM, not all memory reads and writes are saved in the semantic read/write sets. Instead, only those reads and writes that affect linearization of the object and consistency of the transaction are saved. This avoids false conflicts (i.e., conflicts that occur on memory when there is no semantic conflict and there is no need to abort), which degrade the performance of several STM-based data structures. An example is `add` operation in a set. Post conditions will be whether is successful or unsuccessful. Preconditions include only the nodes that will be modified to add the new item (more details about optimistic boosted set operations and how they use semantic read/write sets are shown in Section 3.2.1).

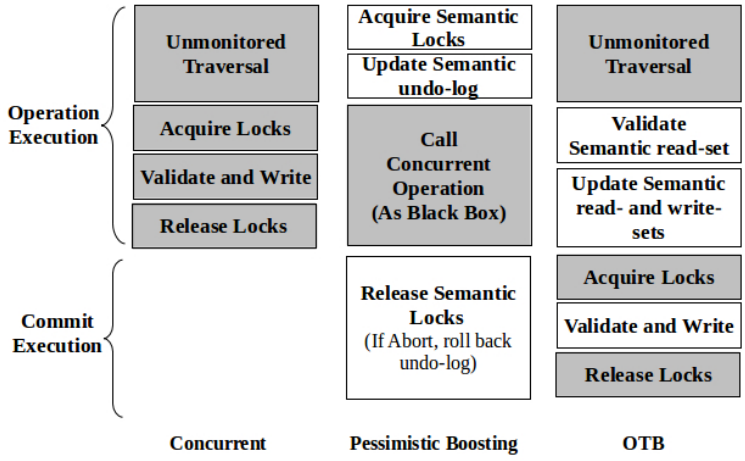


Figure 3.1: Flow of operation execution in: STM, concurrent (lock-based or lock-free) data structures, pessimistic boosting, and optimistic boosting.

Figure 3.1 shows the general structure of concurrent operations in: STM, concurrent (lock-based or lock-free) data structures, pessimistic boosting, and optimistic boosting. The figure provides several key insights. STM provides strong correctness and progress guarantees but at the cost of monitoring each memory access, resulting in performance degradation. Concurrent (non-transactional) data structures have high performance because they only acquire locks (or use CAS operations in case of lock-free objects) at late phases. However, concurrent data structures do not support transactional access or composable operations, which is the main goal of both pessimistic and optimistic boosting. Pessimistic boosting adds

transactional access capabilities to concurrent data structures, but at the cost of pessimistic locking and at the risk of breaking opacity. Optimistic boosting combines the benefits of all these approaches by providing the same methodology of STM at the semantic layer: it (still) traverses the data structure without monitoring (gaining the same benefits of concurrent data structures). Subsequently, it (only) validates its semantic read-set and updates its semantic write-set (gaining the same benefits of pessimistic boosting), and defers any physical modifications of the object to commit time (gaining the same benefits of STM structures).

The optimistic boosting approach thus clearly combines the benefits of the previous approaches: *i)* we provide consistent, atomic, and isolated transactional objects, *ii)* we do not track every read and write like regular STMs, and *iii)* we are optimistic, unlike the original (pessimistic) boosting. These benefits are the direct result of not relying on either the concurrent object or the STM framework as black boxes, like the previous approaches.

Pessimistic and optimistic boosting fundamentally differ in the way concurrent data structures are used. Pessimistic boosting treats concurrent objects as black boxes to synchronize commutable operations at the memory level. This approach, although is completely transparent and decoupled from the underlying data structure, presents oppositionist for more optimizations on these underlying data structures according to their new transactional characteristics. Optimistic boosting, conversely, treats concurrent data structures as white boxes. In fact, it provides a new transactional version of the data structure, based on the same idea of the corresponding concurrent version, and provides the same APIs. In Section 3.2, we show how we optimize each type of data structure (set and priority queue) using optimistic boosting.

The two approaches also diverge in the semantic requirements needed for enabling boosting on a given data structure. Optimistic boosting, unlike pessimistic boosting, does not require well defined commutativity rules or inverse operations. Instead of inserting inverse operations in semantic undo logs, optimistic boosting uses semantic redo logs to publish deferred operations at commit. In data structures like priority queue, inverse operations can be natively un-supported. For example, the inverse of the `add` operation is the removal of an element. This operation is not natively supported in priority queue, because the only permitted `remove` operation is the removal of the minimum element.

3.2 Boosted Data Structures

We now introduce two types of optimistically boosted data structures: set and priority queue. These were specifically chosen as they represent two data structure categories:

- *Commutable Objects.* In set, operations are commutative at the level of keys themselves. 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 3.1, with slight modifications to cope with the different levels of commutativity. Considering this classification, our proposal is general enough to be considered as a guideline for boosting new data structures that are semantically similar to those analyzed in this chapter. In Chapter 7, we propose designing OTB maps and balanced trees in our pot-preliminary work.

3.2.1 Set

Sets are collections of items, which have three basic operations: `add`, `remove`, and `contains`, with the familiar meanings. 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 query 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 [35] 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 references to skip the deleted node. This flag enables skipping a chain of deleted nodes and from returning an incorrect result. It is important to note that, the `contains` operation is wait-free and is not blocked by any writing operation.

Lazy skip-list is more efficient than linked-list as it takes logarithmic time to traverse the set. In skip-list, each node is linked into a subset of the lists, 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 an unified ascending order of levels to avoid deadlocks), instead of locking one pair of nodes as in linked-list. For `add` operation, a $fullyLinkd$ flag is added to each node to logically add it to the set after all levels have been successfully linked.

As shown in Figure 3.1, the pessimistic boosting implementation of set is straightforward

and does not change if the set implementation itself changes (more details are in [37]). 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.

Linked List Implementation

Following the first OTB rule mentioned in Section 3.1, we divide OTB linked-list operations into three step. **Traversal** step is only used to reach the involved nodes, without any addition to the semantic read/write sets. **Validation** step is used to guarantee the consistency of the transaction and the linearization of the list. Post-validation routine is called after each operation, and commit-time-validation is called at commit time and after acquiring the semantic locks. **Commit** step, which writes on the shared list, is deferred to transaction's commit. Following the second rule, we show how we can make specific optimizations on the new transactional linked-list by using the underlying lazy linked-list as a white box.

Similar to lazy linked-list, each operation 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 (sentinel nodes are added as head and tail of the list to handle special cases). To save needed information about these nodes, we use the same STM concepts of read-sets and write-sets, but at the semantic level. In particular, each read-set entry contains the two involved nodes in the operation and the type of the operation. Each write-set entry contains the same items, and also includes the new value to be added in case of a successful `add` operation. The `add` and `remove` operations are not necessarily considered as writing operations, because duplicated items are not allowed in the set. This means that, both `contains` and unsuccessful `add/remove` operations are considered as read operations (which just add entries to the semantic read-set). Only successful `add` and `remove` operations are considered read/write operations (which add entries to both the read-set and the write-set).

Algorithm 1 shows the pseudo code of the linked-list operations. We can isolate four parts of each operation:

- **Local writes check** (lines 2-15). Since writes are buffered and deferred to the commit phase, this step guarantees consistency of further reads and writes. 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. Furthermore, if a transaction adds an item and then removes the same item, or vice versa, operations locally eliminate each other (lines 8 and 14). This elimination can further improve optimistic boosting's performance. Elimination only removes the entries from the write-set and leaves the read-set entries as they are, to maintain transaction isolation by validating the eliminated operations at commit.
- **Traversal** (lines 16-19). This step is exactly the same as in lazy linked-list. It saves

Algorithm 1 Linked-list: `add`, `remove`, and `contains` operations.

```
1:  procedure OPERATION(x)
                                                  ▷ Step 1: search local write-sets
2:      if x ∈ 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 ∈ 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
                                                  ▷ Step 2: Traversal
16:     pred = head and curr = head.next
17:     while curr.item < x do
18:         pred = curr
19:         curr = curr.nerxt
                                                  ▷ Step 3: Post Validation
20:     if ¬ post-validate(read-set) then
21:         ABORT
                                                  ▷ Step 4: Save reads and writes
22:     Compare curr.item with x and check curr.deleted
23:     if Successful add/remove then
24:         read-set.add(new ReadSetEntry(pred, curr, operation))
25:         write-set.add(new WriteSetEntry(pred, curr, operation, x))
26:         return true
27:     else if Successful contains then
28:         read-set.add(new ReadSetEntry(pred, curr, operation))
29:         return true
30:     else if Unsuccessful operation then
31:         read-set.add(new ReadSetEntry(pred, curr, operation))
32:         return false
33: end procedure
```

the overhead of all unnecessary monitoring during traversal that otherwise would be incurred with a native STM algorithm for managing concurrency.

- **Post-Validation** (lines 20-21). At the end of the traversal step, the involved nodes are found in local variables (i.e., *pred* and *curr*). At this point, to avoid breaking opacity [28], the read-set must be post-validated to ensure that the transaction does not see an inconsistent snapshot. The same post-validation mechanism is used in memory-level STM algorithms such as NOrec [19].
- **Saving reads and writes** (lines 22-32). At this point, the transaction can decide on the return value of the operation (line 22). Then, it modifies its read and write sets. Although all operations must add the appropriate read-set entry, only the successful `add/remove` operations modify the write-set (line 25). There is no need to acquire locks for `contains` and unsuccessful `add/remove` operations during commit phase. This way, we keep the *contains* operation lock-free, like lazy linked-list, which allows better performance. Pessimistic boosting, on the other hand, has to acquire semantic locks even for the *contains* operation to maintain consistency and opacity.

The post-validation step is shown in Algorithm 2. 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* (line 11). In case of successful `contains` and unsuccessful `add`, a simpler validation is used. In these particular cases, the transaction only needs to check that *curr* is still not deleted (line 8), 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

Algorithm 2 Linked-list: post 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 successful contains or unsuccessful add then
8:              if curr.deleted then
9:                  return false
10:         else
11:             if pred.deleted or curr.deleted or pred.next
    ≠ curr then
12:                 return false
13:         return true
14:     for all entries in read-sets do
15:         check snapshot of involved locks
16:         if version of one involved lock is changed then
17:             return false
18: end procedure
```

conflicts on *pred* are not real semantic conflicts.

To maintain isolation, a transaction ensures that 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 validation, a transaction takes a snapshot of the locks and makes sure that they are unlocked (lines 2-5). After validation, it ensures that the lock versions are still the same (lines 14-17).

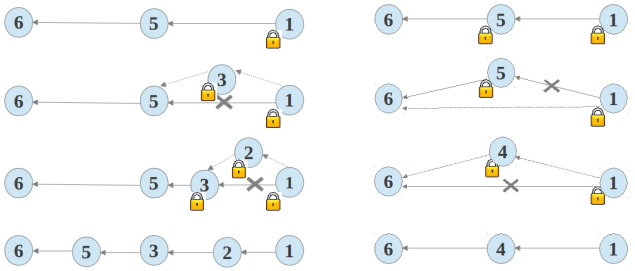
Algorithm 3 Linked-list: commit.

```
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 (or next 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:     else                                          ▷ remove
21:         curr.deleted = true
22:         pred.next = curr.next
23:     for all entries in write-sets do
24:         unlock pred and curr (and next if remove)
25: end procedure
```

Algorithm 3 shows the commit step of the optimistically boosted linked-list. Read-only transactions have nothing to do during commit, because of the incremental validation during the execution of the transaction. For write transactions, the appropriate locks are first acquired using CAS operations. To avoid deadlock, any failure in CAS results in transaction abort and retry (releasing all previously acquired locks). Then validation is called to ensure consistency (lines 2-8). Here, two optimizations are used. The first is the same used by 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 [35]. Second, commit-time-validation does not need to validate lock versions, unlike post-validation, because all nodes are already locked by the transaction.

The next step of commit is to publish writes on the shared linked-list, and then release the locks. This step is not trivial, because each node may be involved in more than one operation

in the same transaction. In these cases, the saved *pred* and *curr* of these operations may change according to which operation commits first. For example, in Figure 3.2(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 saves node 1 as the predecessor of node 3.



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

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

Figure 3.2: Executing more than one operation that involves the same node in the same transaction.

To ensure that operations in this case are executed correctly, three guideline points are followed (described in Algorithm 3):

1. Inserted nodes are locked until the whole commit procedure is finished. Then they are unlocked along with the other *pred* and *curr* nodes (line 17).

2. The items are added/removed in descending order of their values, regardless of their order in the transaction execution (line 9). This guarantees that each operation starts the commit phase from a valid non-deleted node.

3. Operations resume traversal from the saved *pred* to the new *pred* and *curr* nodes (which can be only changed locally, otherwise the transaction will abort). Lines 11-14 encapsulate the logic for this case.

Using these three points, the issue in Figure 3.2(a) will be solved. According to the first point, all nodes (1, 2, 3, 5) are locked and no transaction can access them until commit finished (any transaction will abort if it tries to access these nodes). The second point enforces that node 3 is inserted first. Subsequently, according to the third point, when 2 is inserted, the transaction will resume its traversal from node 1 (which is guaranteed to be locked and non-deleted). It will then detect that node 3 is its new *succ*, and will correctly link node 2.

The removal case is shown in Figure 3.2(b), in which node 5 is removed and node 4 is inserted. Again, 5 must be removed 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. The same procedure holds for the case of the two subsequent `remove` operations.

If the commit-validation fails, or the locks are detected to be acquired by other transactions, any previously locked nodes are released, the local semantic read-set and write-set are cleared, and then abort is called.

Skip List Implementation

Skip-list is a logarithmic data structure, which allows fast set traversals. It is also more suited for optimistic operations than linked-list, because optimism is based on the premise that conflicts are not the common case and the overhead of rolling back (compared to execution) is minimal. For a long linked-list, even if aborts are rare, their effect includes re-traversing the whole list again, to ensure that the linked-list is still consistent after abort. In a skip-list, even if conflicts are greater, the cost of re-traversal is low, which minimizes the overhead of aborts.

Despite the significant improvement in traversal cost and abort overhead, implementation of optimistic boosted skip-list is almost the same as that for the linked-list. Due to space constraints, we only briefly describe the main differences between skip-list and linked-list:

- In skip-list, the read-set and write-set entries contain the entire *pred* and *curr* array, instead of a single pair. At commit time, *pred* and *curr* nodes at all levels are locked and validated.
- We optimize validation in a similar way to what we did for linked-list. We make use of the fact that all items have to appear in the lowest level of the skip-list. For successful *contains* and unsuccessful *add*, it is sufficient to validate that *curr* in the last level 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 have to be validated to prevent conflicts.
- If the involved *curr* node of any operation is not yet $fullyLinked$, it means that another transaction is modifying the node in its commit handler. In this case, the operation waits until the node is $fullyLinked$, and then it proceeds.
- As we mentioned for linked-list, if a node is involved in two operations of the same transaction, the *pred* and *curr* nodes of these operations may change during commit. Thus, we use the same 3-step procedure described in linked-list. The only difference here is that we have more than one level, and each level is independent of the other (which means that *pred* in one level may change while it is still the same in another level). For this case, operations continue traversing to the new *pred* and *curr* for each level separately.

Features

Optimistic boosted set preserves the performance gains of lazy linked-list and skip-list, because it traverses the list without storing in the private read- and write-set all the memory addresses accessed. It also easily integrates with an STM framework, because it handles the involved nodes using semantic read-sets and write-sets. We summarize the advantages of optimistic boosting against its pessimistic version:

- Only the successful write operations acquire locks at commit time. There is no need to acquire locks for `contains` and unsuccessful `add/remove` operations. This way, we keep the *contains* operation lock-free, like lazy linked-list, which allows better performance. Pessimistic boosting, on the other hand, has to acquire semantic locks even for the *contains* operation to maintain consistency and opacity.
- Write operations are atomically published at commit time, which guarantees isolation at both semantic and memory levels.
- Validation is optimized for each operation. Only the involved nodes and links are validated.
- Local operations can eliminate each other, which saves the overhead of unnecessary writes on the shared object. An example is when two subsequent successful `add` and `remove` operations on the same item are performed in the same transaction.
- Lock granularity at commit time can be adapted according to the underlying memory-level STM algorithm. For example, with STM algorithms that use coarse-grained locks such as TML [66] and NOrec [19], the boosted objects can use the same global lock to synchronize their operations at commit (replacing lines 2-8 in Algorithm 3). This saves the unnecessary overhead of managing fine-grained locks. Adapting the commit procedure at run-time according to the used STM algorithm is straightforward and does not incur extra overhead.

The optimistic boosting methodology enables these optimizations because of two reasons. First, the underlying concurrent data structure is not considered as a black box. Instead, it is specifically optimized for enhancing performance. Second, optimistic boosting follows the same STM approach for validation and synchronization, which enables easy integration with STM algorithms.

Correctness

Although optimistic boosting does not use the lazy set (using either linked-list or skip-list) as a black box, it uses the same mechanisms for traversing the set and validating the nodes. For this reason, we can assess the correctness of our set implementations by relying on the correctness of the lazy implementations of the concurrent set. The correctness of the optimistic boosted set can be proved in the following two steps. The first step is to show that each single operation is still linearizable, like the lazy set. The second step consists of

showing the correctness and progress guarantees of the transaction as a whole.

Each operation traverses the set exactly like in the lazy set. Then, instead of acquiring locks on the involved nodes instantaneously, the operations acquire the same locks at commit time. Since the same validation is done after lock acquisition, the linearization points of all the operations are exactly the same as in lazy set. This means that lock acquisition is just shifted forward and gathered at commit time (for brevity, we do not discuss the linearization points of lazy linked-list and lazy skip-list; see [35] for details). The optimized validation of successful `contains` and unsuccessful `add` can be easily shown to be correct, because these operations only care about the existence of the item during commit, as they are not affected by *pred*. Finally, reordering the execution of the operations in the descending order of item values during commit does not affect linearization. This is because, two-phase locking and validation of locks during the post-validate procedure guarantee that all operations are atomically – all or nothing – published on the shared list.

Considering a transaction as a whole, the combination of lazy writes and post-validation guarantee consistency. The same approach is used at memory level in many STM algorithms such as NOrec [19] to guarantee opacity. Specifically, when a transaction starts to publish its write-set at commit, it is guaranteed that all operations in its read-set are still valid, and that no other transaction is able to access the involved nodes until commit is finished. Unlike lazy set, unsuccessful `contains` and `remove` need to be validated in order to ensure that the item does not exist and the returned value is valid. Searching in the local write-sets before traversing the shared list maintains the effect of lazy writes. Elimination of the operations saves unnecessary overhead at commit, without affecting the correctness of the whole transaction, because eliminated operations are not removed from the read-set. Atomicity and isolation are guaranteed by the two-phase locking mechanism at commit time. Two-phase locking also guarantees deadlock-freedom.

3.2.2 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 [39]. 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 4 shows the flow of pessimistic boosted operations (more details are in [37]). 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 4 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, the guidelines of optimistic boosting can also help to implement a semi-optimistic heap-based priority queue, as we will show in the next section.

Semi-Optimistic Heap Implementation

A semi-optimistic implementation of a heap-based priority queue is achieved by using the following three optimizations on the original pessimistic boosting implementation (these optimizations are illustrated in Algorithm 5):

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 5 Optimistic boosted heap-based priority queue.

```
 1: procedure ADD(x)
 2:     if lock holder then
 3:         PQ.add(x)
 4:     else
 5:         redo-log.append(x)
 6: end procedure

 7: procedure REMOVEMIN
 8:     Lock.acquire
 9:     for entries in redo-log do
10:         PQ.add(entry.item)
11:     x = PQ.removeMin()
12: end procedure
```

The same idea of our enhancements has been used before in the TML algorithm [66] 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.

Skip-List Implementation

In this version, optimistic boosted priority queue wraps the same optimistic boosted skip-list described in Section 3.2.1. 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 [39]).

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 6 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

Algorithm 6 Optimistic boosted skip-list-based priority queue.

```
 1: procedure ADD(x)
 2:     if skipList.add(x) then
 3:         localPQ.add(x)
 4:         return true
 5:     else
 6:         return false
 7: end procedure

 8: procedure REMOVEMIN
 9:     localMin = localPQ.getMin()
10:     sharedMin = lastRemovedMin.next[0]
11:     if localMin ¡ sharedMin then
12:         if ¬ skipList.contains(sharedMin) then
13:             Abort
14:         if lastRemovedMin.next[0] ≠ sharedMin then
15:             Abort
16:         return localPQ.removeMin()
17:     else
18:         if ¬ skipList.remove(sharedMin) then
19:             Abort
20:         if lastRemovedMin.next[0] ≠ sharedMin then
21:             Abort
22:         lastRemovedMin = sharedMin
23:         return sharedMin
24: end procedure
```

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

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.

3.3 Evaluation

We now evaluate the performance of both optimistic boosted set and priority queue. In each experiment, the number of adds and removes are kept equal to ensure the same structure size during the experiments. 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 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. The machine runs Ubuntu Linux 10.04 LTS 64-bit.

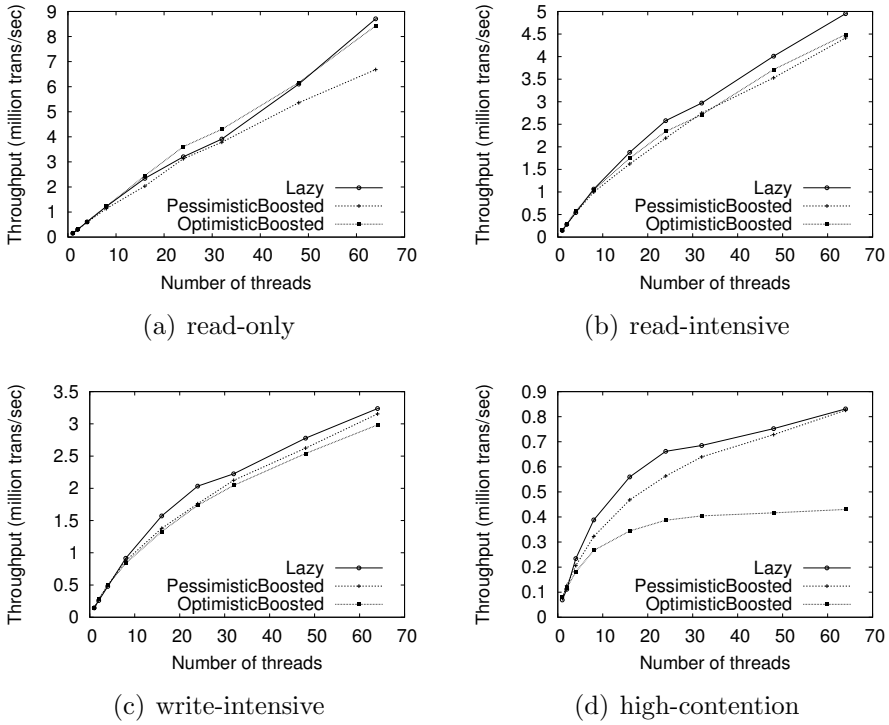
We use transactional throughput as our key performance indicator. Although abort rate is another important parameter to measure and analyze, it is meaningless in our case. Both lazy and pessimistic boosted sets do not explicitly abort the transaction. However, there is an internal retry for each operation if validation of the involved nodes fails. Additionally, pessimistic boosting aborts only if it fails to acquire semantic locks, which is less frequent than validation failures in optimistic boosting.

It is worth noting that lazy set does not interact with the transaction at all. We only show it as a rough upper bound of boosting algorithms, but it actually does not support transactional operations.

3.3.1 Set

We compared optimistic boosted set with lazy set (described in [35]) and pessimistic boosted set (described in [37]). In order to make the experiments fair, the percentage of writes in all of the experiments is the percentage of successful ones, because an unsuccessful `add/remove` operation is considered as a read operation. To roughly 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 in a previous transaction as a parameter, to be successful.

We first show the results for a linked-list implementation of the set. In this experiment, we used a linked-list with 512 nodes. Figure 3.3 shows results for 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). In both read-only and read-intensive workloads, optimistic boosting is closer to the optimal lazy list than pessimistic boosting. This is expected, because pessimistic boosting incurs locking overhead even for read operations. In contrast, optimistic boosting, like lazy list, does not acquire locks on read operations, although it still has a small overhead in validating read-sets, which does not exist in lazy list. For the write-intensive workload, pessimistic boosting starts to be slightly better than optimistic boosting, and the gap increases in high contention workloads. This is also expected, because contention becomes very high, which increases abort rate. (recall that



(a) read-only (b) read-intensive

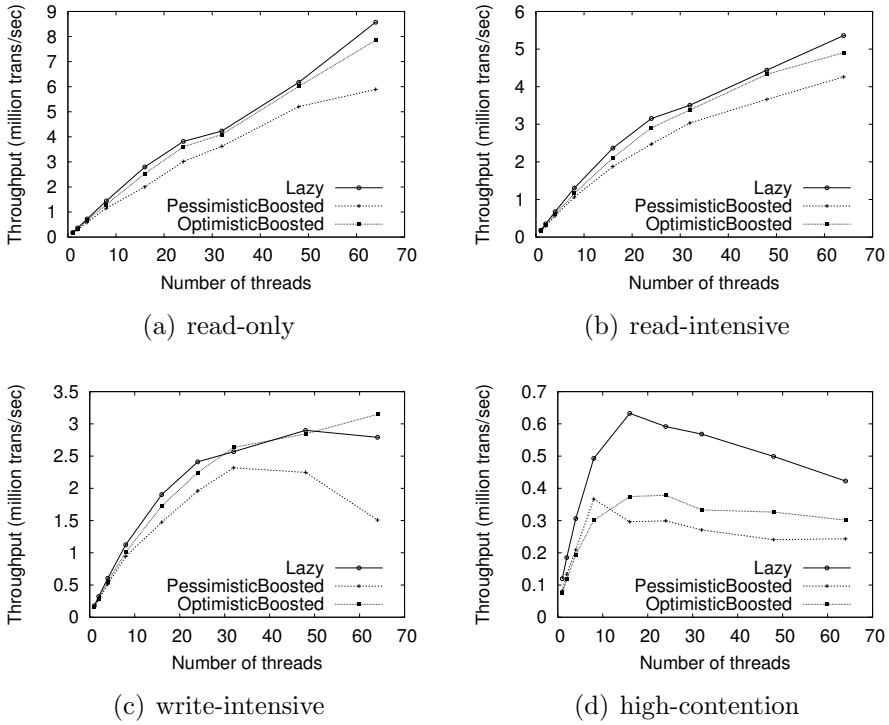
(c) write-intensive (d) high-contention

Figure 3.3: Throughput of linked-list-based set with 512 elements, for 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).

aborts have high overhead due to re-traversing the list in linear time). However, we can conclude that in practical cases, optimistic boosting performs reasonably well. Besides, it offers strong correctness and progress guarantees, and easy integration with STM frameworks.

In Figure 3.4, the same results are shown for a skip-list-based set of the same size (512 nodes). The results show that optimistic boosting is better in all cases, including the high contention case. These results confirm that optimistic boosting gains more from reducing the overhead of aborts. Although contention is almost the same (for a set with 512 nodes, contention is relatively high), using skip-list instead of linked-list enhances optimistic boosting more than pessimistic boosting.

The last experiment, in Figure 3.5, 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 are still practical, optimistic boosting is up to 2x better, even for write-intensive and high contention workloads. This is mainly because, the overhead of eager locking in pessimistic boosting becomes useless, and it is better to use an optimistic algorithm instead of a conservative pessimistic one.



(a) read-only (b) read-intensive

(c) write-intensive (d) high-contention

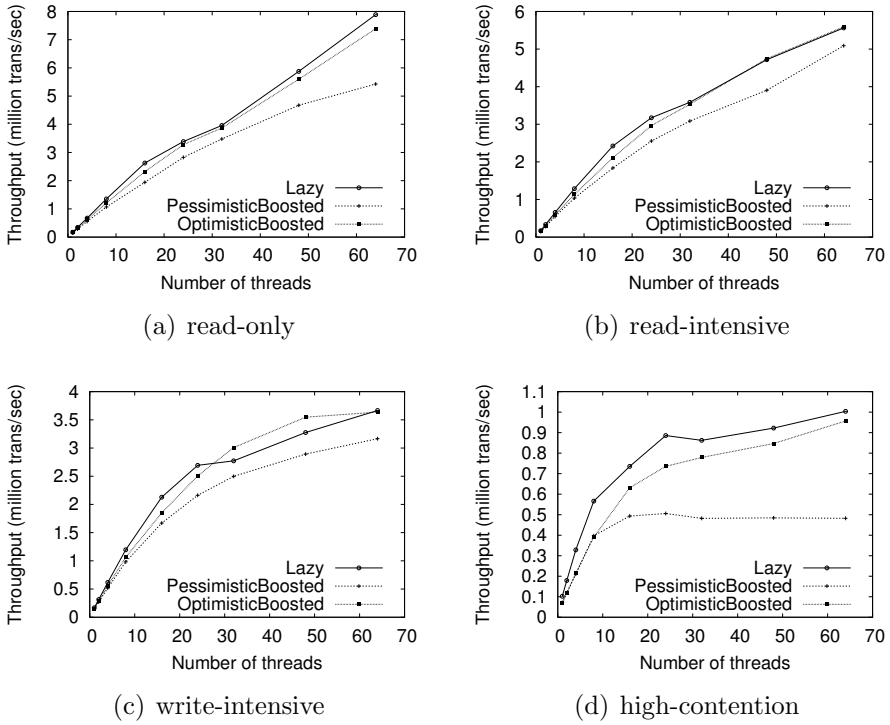
Figure 3.4: Throughput of skip-list-based set with 512 elements, for four different workloads: read-only (0% writes), read-intensive (20% writes), write-intensive (80% writes), and high contention (80% writes and 5 operations per transaction).

3.3.2 Priority Queue

We now show the performance of heap-based priority queue and illustrate how our semi-optimistic implementation enhances performance. We then discuss the skip-list-based priority queue, and compare the performance of pessimistic and optimistic implementations.

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 3.6 illustrates how our three optimizations (described in Section 3.2.2) 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.

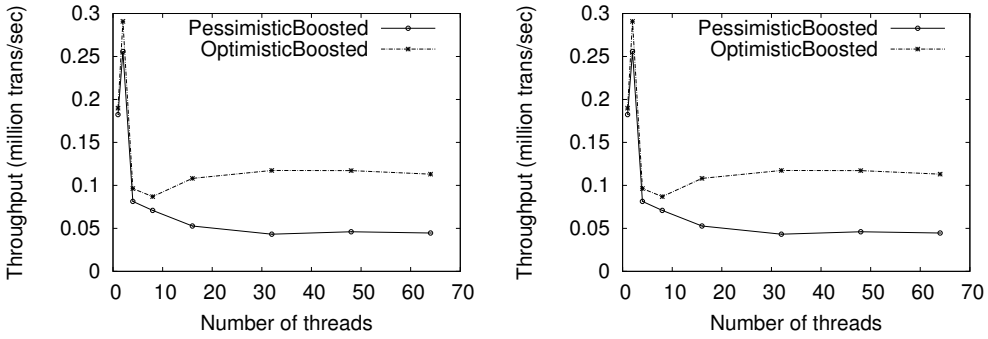
The results show that our semi-optimistic boosting implementation is faster than pessimistic boosting irrespective of the transaction size (1 or 5 operations). Increasing the transaction size to 5 affects the performance of both algorithms, because both acquire global locks when



(a) read-only (b) read-intensive

(c) write-intensive (d) high-contention

Figure 3.5: Throughput of skip-list-based set with 64K elements, for 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).

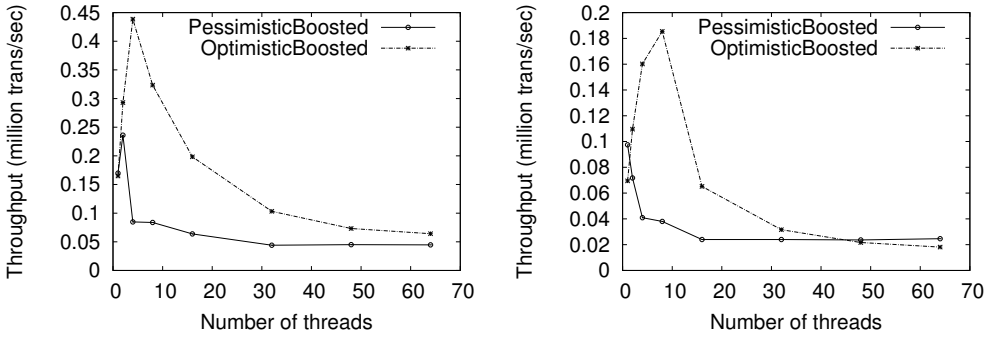


(a) Transaction size = 1 (b) Transaction size = 5

Figure 3.6: Throughput of heap-based priority queue with 512 elements, for two different transaction sizes (1, 5). Operations are 50% add and 50% removeMin.

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 3.2.1)



(a) Transaction size = 1 (b) Transaction size = 5

Figure 3.7: Throughput of skip-list-based priority queue with 512 elements, for two different transaction sizes (1, 5). Operations are 50% add and 50% removeMin.

as the base of priority queue implementation. Figure 3.7 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 3.6 and 3.7, 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 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.

3.4 Summary

We presented *Optimistic Transactional Boosting* as an optimistic methodology to convert concurrent data structures to transactional ones. Transactions use semantic read/write sets to locally save the operations of boosted objects, and defer modifying the shared objects to 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 with the maximum possible optimizations for their new transactional characteristics.

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 4

Integrating OTB with DEUCE Framework

All previous proposals to implement transactional data structures, including boosting, 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 transactionally boosted data structures with the current STM frameworks. One of the main benefits of optimistic boosting (compared to the original *pessimistic* boosting) is that it uses the terms validation and commit in the same way as many STM algorithms [19, 23], but in 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 [44] 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 (described in Chapter 3) in DEUCE framework. We extend two STM algorithms to communicate with OTB-Set (NOrec [19] and TL2 [23]). 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.

4.1 Extension of DEUCE Framework

DEUCE [44] 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 [19], TL2 [23], and LSA [58]). 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.

4.1.1 Programming Model

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 7. In this example, all transactions access a shared set (*set1*), and two shared integers (*n1* and *n2*) which hold the number of successful and unsuccessful add operations on *set1*, respectively. Each thread atomically calls *method1* as a transaction using the *@Atomic* annotation. In *method1*, both set operation (add operation) and traditional memory access (incrementing the shared integers) have to be executed atomically as one block, without breaking consistency, atomicity, or isolation of the transaction.

To execute such an atomic block, all previous proposals use a pure STM-based implementation of *set1*, to allow STM frameworks to instrument the whole transaction. Our extended framework, conversely, is the first proposal that uses a more efficient (transactionally boosted) implementation of *set1*, rather than an inefficient STM implementation, and at the same time allows an atomic execution of this kind of transaction.

Algorithm 7 An example of using Atomic blocks in the new DEUCE framework.

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

4:  @Atomic
5:  procedure METHOD1(x)
6:      if set1.add(x) == true then
7:          n1++
8:      else
9:          n2++
10: end procedure
```

4.1.2 Framework Design

Figure 4.1 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 4.1.2 the original building blocks of the DEUCE framework (the white blocks with numbers 1-3). Then, in Section 4.1.2, 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 [44].

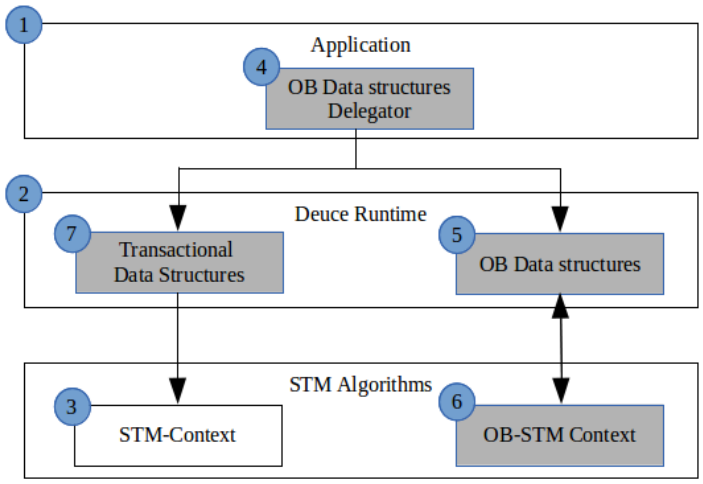


Figure 4.1: New design of DEUCE framework.

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 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 [44].

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*.

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 7, 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 [32]. As a case study, in Section 4.2.1, 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 4.1, 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 4.2, OTB-Set interacts with NOrec [19] and TL2 [23] 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 4.1.2, 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 4.2, 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.

4.2 Case Study: Linked List-Based Set

Following the framework design in Section 4.1, we show a case study on how to integrate an optimistically boosted version of a linked-list-based set in the modified DEUCE framework[1]. This is done using the following two steps:

- Modifying OTB-Set implementation (described in Section 3.2.1) 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 4.1, we will need to implement a new OTB-STM context for both algorithms[2].

4.2.1 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:

- *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:

[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 4.1), but they will use an STM-based implementation of the set rather than our optimized OTB-Set.

- *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 [35].
- *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.

4.2.2 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 [28]. 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 1). 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.

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 4.1.2, because there is no use of the semantic locks

[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.

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.

4.2.3 Integration with TL2

TL2 [23], 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.

4.3 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.

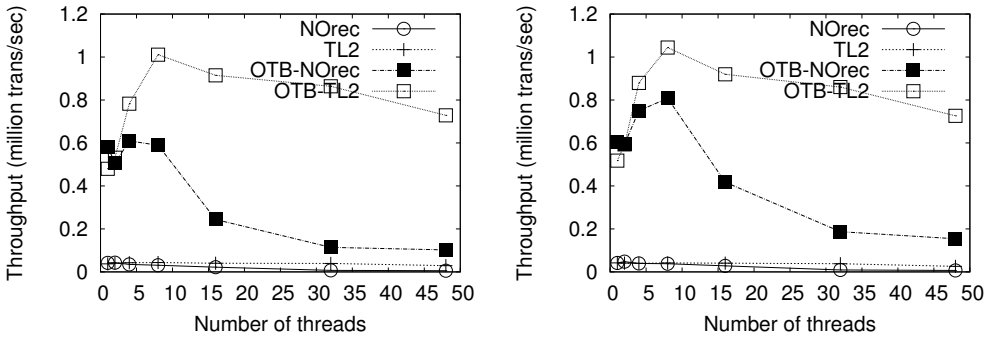
In each experiment, we compare the modified OTB-NOrec and OTB-TL2 algorithms (which are internally calling OTB-Set operations) with the traditional NOrec and TL2 algorithms (which internally call a pure STM version of the set).

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 7, 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.

4.3.1 Linked-List Micro-Benchmark

Figure 4.2 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

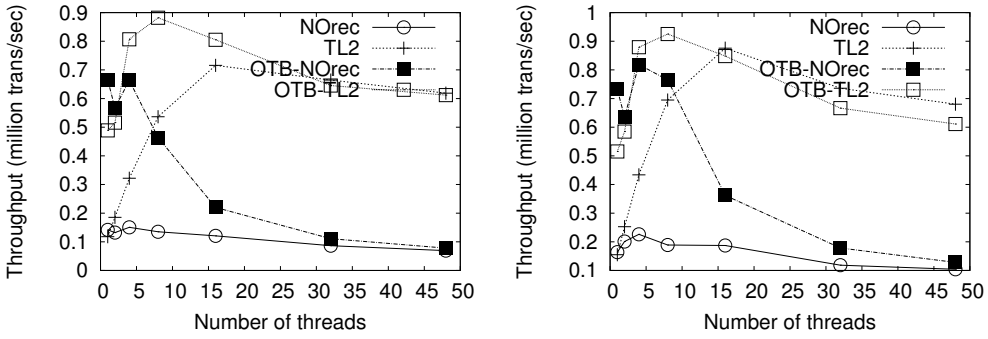
Figure 4.2: Throughput of linked-list-based set with 512 elements, for two different workloads.

4.3.2 Skip-List Micro-Benchmark

Results for skip-list are shown in Figure 4.3. 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.

4.3.3 Integration Test Case

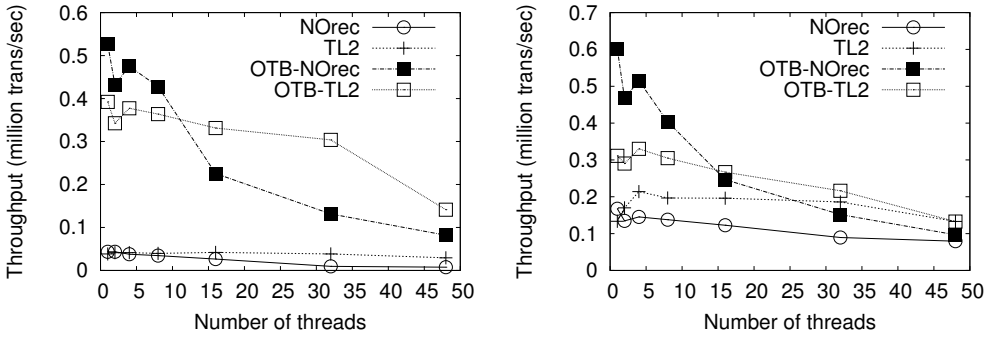
In this benchmark, we have six shared variables in addition to the shared OTB-Set (number of successful/unsuccessful adds, removes and contains). Each transaction executes a set operation (50% reads) and then it increments one of these six variables according to the type of the operation and its return value. As all transactions are now executing writes on few memory locations, contention increases and performance degrades on all algorithms.



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

Figure 4.3: Throughput of skip-list-based set with 4K elements, for two different workloads.

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 4.4: Throughput of Algorithm 7 (a test case for integrating OTB-Set operations with memory reads/writes). Set operations are 50% add/remove and 50% contains.

4.4 Summary

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* (OTB). 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. As a future work, this case study can be generalized with some slight modifications and some relaxations in the STM-algorithm-specific optimizations proposed in this chapter. Performance of micro-benchmarks using the modified framework is improved by up to 10x over the original framework.

Chapter 5

Remote Transaction Commit

we present Remote Transaction Commit (or RTC) – a mechanism for processing commit phases of STM transactions remotely. 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. 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 5.1.2 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 [19]. 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 [64], and a good candidate for hybrid transactional memory approaches [60]. Validation

[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 usage of CAS operations can seriously hamper application performance [39], , especially in multicore architecture.

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 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 5.2, we show RTC's implementation using one main server and one secondary server. Then, in Section 5.5, 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.

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.

5.1 Design

The basic idea of RTC is to execute the commit phase of a transaction in a dedicated main server core, and 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 5.1 shows the structure of a NOrec transaction. A transaction can be viewed as being composed of three main parts: initialization, transaction body, and transaction commit. The initialization part adjusts 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 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 atomic CAS to increase the global timestamp). The write-set is then published in shared memory, and finally, the global lock is released.

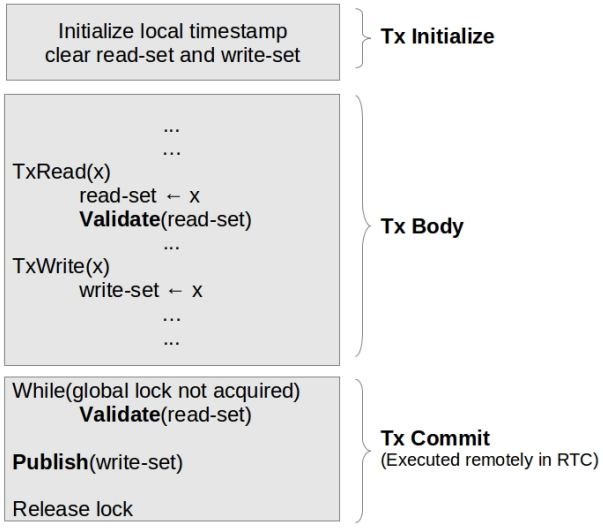


Figure 5.1: Structure of a NOrec transaction

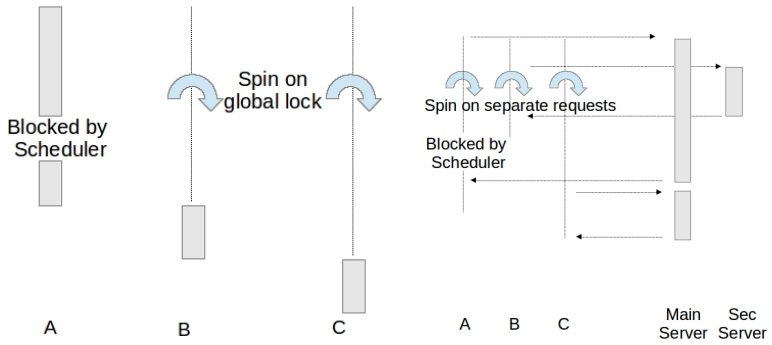
This well defined mechanism of NOrec can be converted to remotely executing the transaction commit part. Unlike lock-based applications, which contains programmer-defined locks with generic critical sections, RTC precisely knows the number of locks (only one global lock), when to execute commit (at transaction end), and what to execute inside commit (validating read-sets and publishing write-sets). This simplifies the role of servers, in contrast to server-based optimizations for locks such as RCL [46] and Flat Combining [36], which need additional mechanisms (either by re-engineering as in RCL or at run-time as in Flat Combining) to indicate to servers what procedures to execute on behalf of the clients.

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 transaction's context (read-sets and write-sets, local timestamp, and bloom filters). This context is bound to the transaction's 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 main server is executing a request, the secondary server also loops on the same array, searching for independent requests. It doesn't execute any client requests unless the main server is executing another non-conflicting request.

Figure 5.2 illustrates the flow of execution in both NOrec and RTC. Assume we have three

transactions. Transaction A is a long running transaction with a large write-set. Transaction B does not conflict with A and can be executed concurrently, while transaction C is conflicting with A. Figure 5.2(a) shows how NOrec executes these three transactions. If A acquires the lock first, then both B and C will spin on the shared lock until 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 A is blocked by the OS scheduler, then both the spinning transactions will also wait until A resumes, paying an additional cost. This possibility of OS blocking increases with the number of busy-waiting transactions.

In Figure 5.2(b), RTC moves the execution of A's commit to the main server. Transactions B and 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 A's execution, the secondary server (which is dedicated to detecting dependency) discovers that B can run concurrently with A, so it starts executing B without waiting for A to finish. Moreover, when A is blocked by the OS scheduler, this blocking does not affect the execution of its commit on server, and does not block other transactions. Blocking of the servers is much less frequent here, because no client transactions are allowed to execute on the server cores.



(a) NOrec (b) RTC

Figure 5.2: Flow of commit execution in NOrec and RTC

5.1.1 Dependency Detection

RTC leverges on a secondary server to solve the problem of unnecessary serialization of independent commit requests. The secondary server uses bloom filters [9] 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's writes, and the second one is a read-write filter which represents the union of the transaction's reads and writes. If the read-write filter of a transaction X does not intersect with the write filter of the transaction Y currently ex-

ecuted in the main server, then it is safe to execute X in the secondary server. (We provide a proof of the independence between X and Y in Section 5.3).

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 5.2 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 with the main server. This overhead will be relatively small if commit phase is long, and transactions are independent. On the other hand, if write-sets are short (indicating short commit phase), 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 execution before the secondary server makes real 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 5.5, 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 is the bloom filter size. If it is too small, many false conflicts will occur and the detector will not be efficient. On the other hand, large bloom filters need large constant time to be accessed. Bloom filter access must be fast enough to be useful. In our experiments, we used the same size as in RSTM's default configuration (1024 bits), as we found that other sizes gives similar or worse performance.

5.1.2 Analysis of NOrec Commit Time

As we mentioned before, RTC is more effective if the commit phase is not too short. Table 5.1 provides an analysis of NOrec's commit time ratio of the STAMP benchmarks [53][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

[3]We excluded both yada and bayes applications of STAMP here and in all our further experiments as they evidenced errors (segmentation fault) when we tested them on RSTM, even in the already existing algorithms like NOrec.

that the non-transactional part 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 5.1: Ratio of NOrec's commit time in STAMP benchmarks

As our experimental results show in Section 5.4, increase in RTC's performance is proportional to the commit time. In fact, benchmarks with higher percentage of commit time (genome, ssca, 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 almost read-only workloads (labyrinth) show no difference between NOrec and RTC because nothing is done in read-only transaction's commit.

5.2 RTC Algorithm

The main routines of RTC are servers loops and the new commit procedure. The initialization procedure and transaction body code are straightforward, so we only briefly discuss them. Our full implementation, examples and test-scripts used for experimental evaluation, are available for reviewers as a supplemental material.

5.2.1 RTC Clients

Clients' commit requests are triggered using a cache-aligned requests array. Each commit request contains three items (shown in Figure 5.3):

- **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 a reference to a client's transaction 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 servers. The second is to bind the servers to their reserved cores.

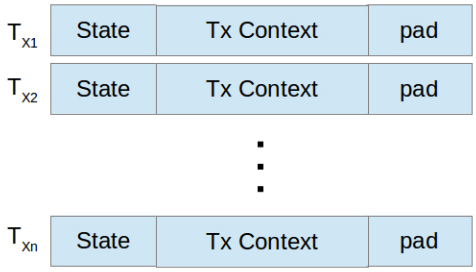


Figure 5.3: RTC's Cache-aligned requests array

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 8 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, 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 main server is the only thread which changes the timestamp.

Algorithm 9 shows the commit procedure of RTC's clients. Read-only transactions do not need to acquire any locks and their commit phase is straightforward. A read-write transaction starts commit by validating its read-set to reduce the overhead on servers if it's already invalidated (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 reuse (line 13).

Algorithm 8 RTC: client validation

```
 1: procedure CLINET-VALIDATION
 2:     t = global_timestamp
 3:     if t is odd then
 4:         retry validation
 5:     Validate read-set values
 6:     if t ≠ global_timestamp then
 7:         retry validation
 8:     else
 9:         return t
10: end procedure
```

Algorithm 9 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 ∉ (READY, ABORTED)
 9:         if req.state = ABORTED then
10:             TxAbort()
11:         else
12:             TxCommit()
13:         ResetFilters()
14: end procedure
```

5.2.2 Main Server

The main server is responsible for executing the commit part of any pending transaction. Algorithm 10 shows the main server loop. By default, 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's 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's 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.

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 26–28) to prevent the main server from proceeding until the

secondary server finishes its work. As we will show in the correctness part (in Section 5.3), this *servers_lock* gurantees that the secondary server will only execute as an extention to the execution of the main server. In 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.

Algorithm 10 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 ← 1, num_transactions do
 5:             req ← req_array[i]
 6:             if req.state = PENDING then
 7:                 if ¬ Server-Validate(req.Tx) then
 8:                     req.state = ABORTED
 9:                 else if write_set_size < 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)
17:     global_timestamp++
18:     WriteInMemory(req.Tx.writes)
19:     global_timestamp++
20:     req.state = READY
21: end procedure

22: procedure Commit(DD-Enabled)
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:     servers_lock = false
29:     req.state = READY
30: end procedure
```

5.2.3 Secondary Server

Algorithm 11 shows the secondary server's operation. It loops almost the same as 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).

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 after acquiring *servers_lock* that the global timestamp is not changed by the main server (line 16). This means that even if the secondary server reads any false information from the above three points, it will detect that soon after it acquires *servers_lock* when it sees a different timestamp. The next step is to either commit or abort (similar to main server) 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 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 in the beginning and 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*).

Algorithm 11 RTC: secondary server

```
 1: procedure SECONDARY SERVER LOOP
 2:     while true do
 3:         for i ← 1, num_transactions do
 4:             if DD = false then continue
 5:             s = global_timestamp
 6:             if s&1 = 0 then continue
 7:             req ← 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
13:     if ¬ Server-Validate(req.Tx) then
14:         aborted = true
15:     if CAS(servers_lock, false, true) then
16:         if s <> global_timestamp 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
```

5.3 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. We then 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's correctness is similar to that of NOrec. Briefly, post validation in Algorithm 8 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 of incrementing 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's 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 RTC's 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 in 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 $rwf(t2) \cap wf(t1) = \emptyset$, t_1 can be linearized 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 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 (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 the timestamp. If the timestamp is 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

[4] In all of the proof arguments, we assume that instructions are executed in the same order as shown in Section 5.2'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).

to search for another request.

More detailed, in lines 26-28 of Algorithm 10, the main server increments the timestamp in a mutually exclusive way with lines 15-24 of the secondary server execution in Algorithm 11 (because both parts are enclosed by acquiring and releasing the *servers_lock*). Following all possible race conditions between line 27 of Algorithm 10 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 5.4 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 11 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 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).

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 5.4 shows that the global timestamp increment (to be odd and then even) encapsulates the

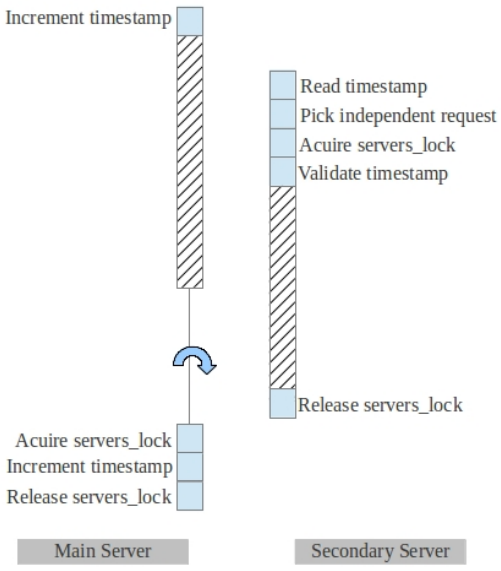


Figure 5.4: Flow of commit execution in the main and secondary servers. Even if the main server finishes execution before the secondary server, it will wait until the secondary server releases the *servers_lock*.

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 time stamp is odd.

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

5.4 Experimental Evaluation

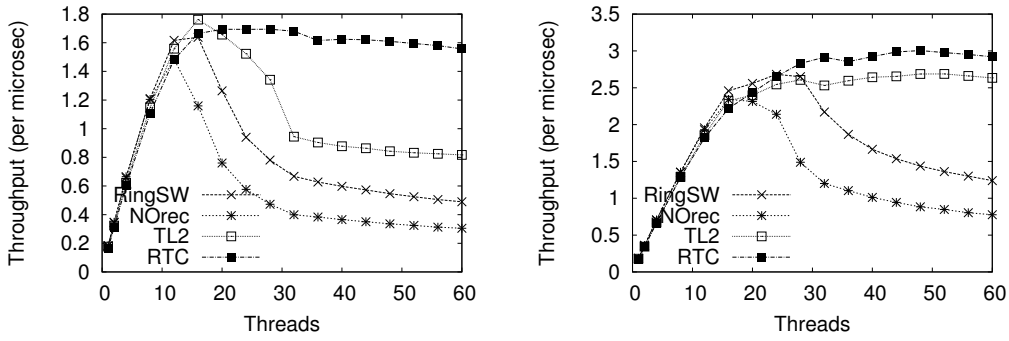
We implemented RTC in C/C++ (compiled with gcc 4.6) and ported to the RSTM framework [48] (compiled using default configurations) for being tested using its interface. Our experiments were performed on a 64-core AMD Opteron machine (128GB RAM, 2.2 GHz).

Our benchmarks for evaluation included micro-benchmarks (such as RB-Tree, linked list, and hash map), and the STAMP benchmark suite [53]. We also evaluated multiprogramming cases, where number of transactions is more than number of cores. Our competitors against RTC included NOrec (representing global metadata approaches), RingSW (representing bloom filter approaches), and TL2 (representing ownership-records based approaches). We used a privatization-safe version of TL2 for a fair comparison. The competitor STM

algorithm and the benchmark implementations used were those available in RSTM. All data points shown are averages of 5 runs.

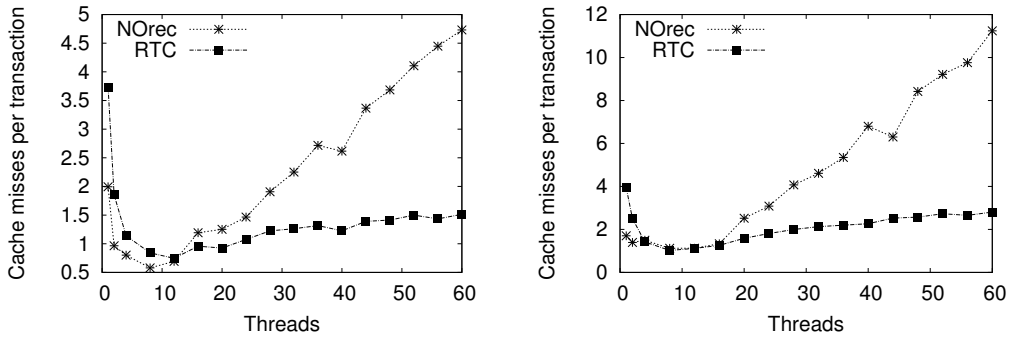
5.4.1 Micro-Benchmarks

Red-Black Tree. Figure 5.5 shows the throughput of RTC and its competitors on red-black tree with 64K elements and a delay of 100 no-ops between transactions. In Figure 5.5(a), when 50% of operations are reads, all algorithms scale similarly, but RTC sustains high throughput, while other algorithms' throughput degrade. This is a direct result of keeping the commit phase isolated in server cores. In Figure 5.5(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 peak performance, 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 5.5: Throughput (per micro-second) on red-black tree with 64K elements

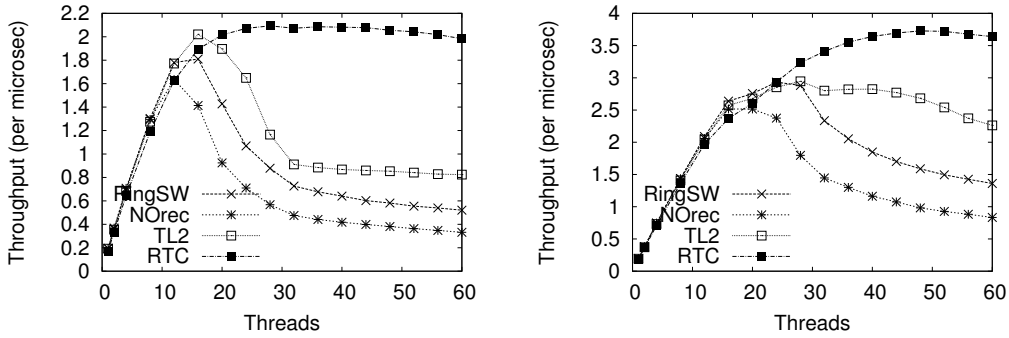


(a) Large tree (64K nodes) (b) Small tree (64 nodes)

Figure 5.6: Cache misses per transaction on red-black tree

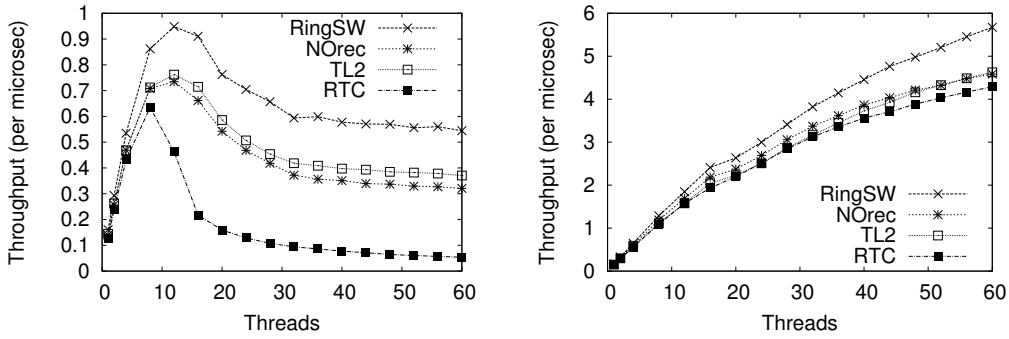
Additionally, we focused on making a connection between the performance (in terms of throughput) and the average number of cache misses per transaction generate by NOrec and RTC. Figure 5.6 shows the results. At high number of threads, the number of cache misses per transaction on NOrec is 5 (2.5× more than RTC) in large trees (which represents low contention), and 10 (5× more than RTC) in small trees (which represents high contention). RTC sustains about 2 cache misses per transaction in both cases, which confirms that with our approach cache misses becomes independent from the contention. Comparing Figures 5.5(a) and 5.6(a), 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's approach.

Hash Map. Figure 5.7 shows the results for the hash map micro-benchmark with 10,000 elements and 256 buckets. Results are similar to the red-black tree benchmark. All algorithms scale the same, but RTC yields higher throughput at large number of threads.



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

Figure 5.7: Throughput on hash map with 10000 elements, 100 no-ops between transactions



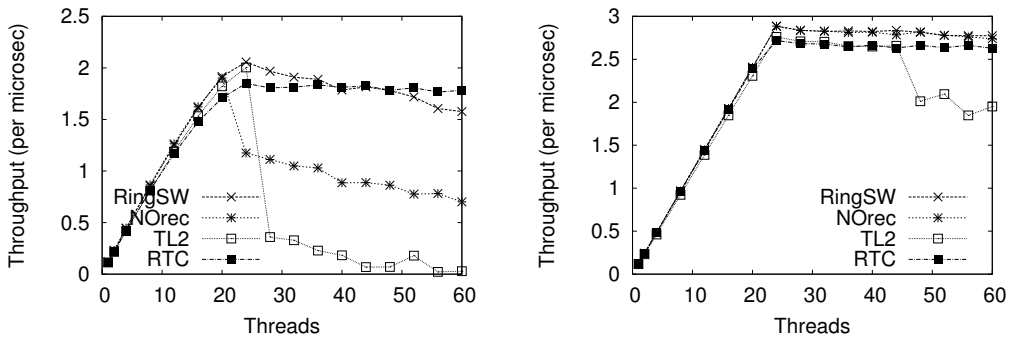
(a) 50% reads (b) 98% reads

Figure 5.8: Throughput on doubly linked list with 500 elements, 100 no-ops between transactions

Linked List. In Figure 5.8(a), we show the worst case for RTC, in order to conduce a fair comparison. Linked List is a benchmark which exposes non optimal characteristics in terms of commit time ratio (in Table 5.1, we show how usually this ratio is, in benchmarks like STAMP). In fact, in a doubly linked list with 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 percentage of commit time is probably less than 1%. Of course, this ratio will decrease more if the size of the linked list increases. Comparing this ratio with ratios in Table 5.1, we can easily understand why RTC is not performing well in this specific case. This issue does not occur when the number of read-only operations is relatively very large (Figure 5.8(b)), 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 [63]. 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.

5.4.2 Performance under Multiprogramming

In this experiment, we created up to 64 threads on a 24-core AMD Opteron machine. RTC should be effective in this case, because servers still have the highest probability to execute with minimal interruption. In other algorithms, the probability of blocking lock holders increases, because contention increases on the available cores.



(a) 50% reads (b) 98% reads

Figure 5.9: Throughput on red-black tree with 64K elements, 100 no-ops between transactions, on 24-core machine

Results in Figure 5.9(a) confirm our hypothesis. In this experiment, 50% of operations are read. After 24 threads, throughput of all algorithms degrade due to multiprogramming. RTC does not suffer from this issue and yields better performance. Degradation of RingSW is slight but it still exists, while TL2 has the worst performance in this case.

In Figure 5.9(b), when 98% of operations are read, RTC servers will not be doing useful work almost all the time. This is why after 24 threads, the overhead of dedicating two cores for the servers begins to manifest. If the number of cores is large enough, the margin between RTC and other algorithms in this case will be relatively small.

5.4.3 STAMP

Figure 5.10 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 5.1. RTC has more impact when commit time is relatively large, especially when the number of threads increases.

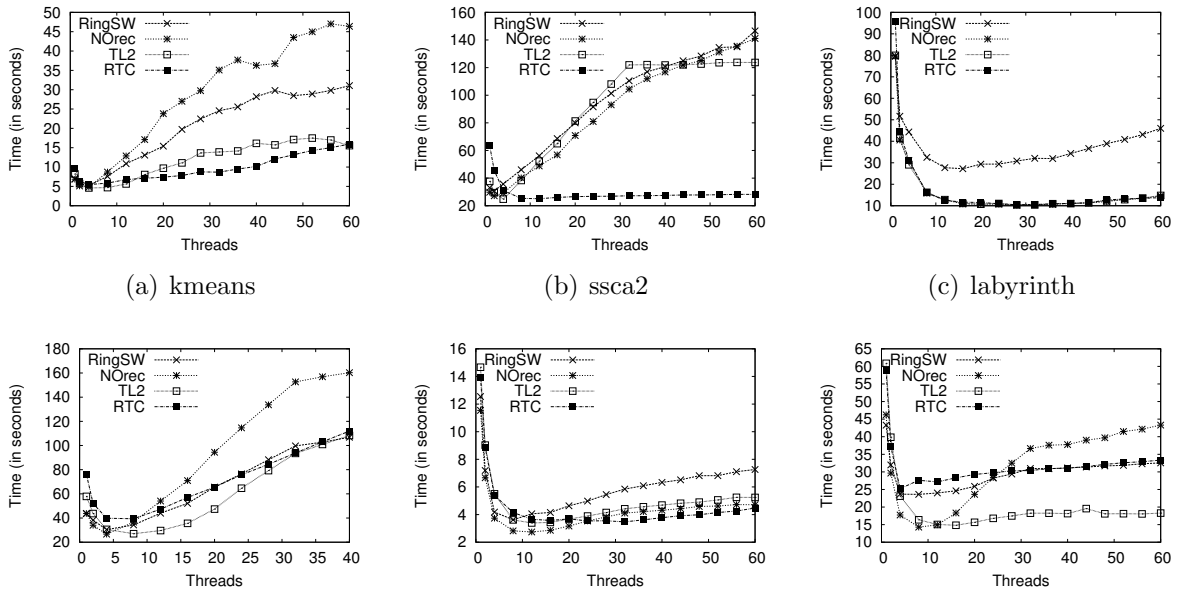
In four out of these six benchmarks (ssca2, kmeans, genome, and labyrinth), RTC has the best performance when number of threads exceeds 20. Also, in all cases, RTC outperforms NOrec at high number of threads. Even for the vacation benchmark, where the commit time percentage is small (6%–50%), RTC outperforms NOrec and has the same performance as RingSW after 24 threads. For kmeans and ssca2, which have the largest commit overhead according to Table 5.1, RTC has better performance than all algorithms at low number of threads. For labyrinth, RTC performs the same as NOrec and TL2, and better than RingSW. This is because, RTC doesn't have any overhead on read-only transactions, which dominate in labyrinth. In all cases, RTC's overhead at low number of threads (less than 8) is more than that of other algorithms. This is reasonable, due to the server-client communication cost. That cost is dominated by RTC's other improvements at high number of threads.

5.5 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.

This is not applied for secondary server. This is because adding more secondary servers (which search for independent requests) 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



(a) kmeans (b) ssca2 (c) labyrinth

(d) intruder (e) genome (f) vacation

Figure 5.10: Execution time on STAMP benchmark

RTC with more secondary servers, which are shown in Figure 5.11. They highlight that the synchronization overhead needed for managing the concurrent execution of more secondary servers, is higher than the gain achieved.

In Figure 5.11, N reads and N writes of a large array's elements are executed in a transaction, 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 5.11 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.

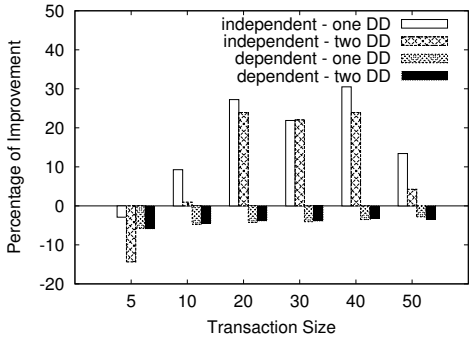


Figure 5.11: Effect of adding dependency detector servers

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.

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 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.

5.6 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. RTC reduces this overhead with a simple idea: execute the commit phase in a dedicated servicing thread. (Independent transactions are detected in another thread, and they are consistently executed concurrently.) 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.

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. Our work also illustrates that, the idea is more

effective for coarse-grained STM algorithms than fine-grained, due to the reduced number of synchronizations that must be optimized. Moreover, it yields additional benefits such as privation-safety and efficient interaction with non-transactional code.

Chapter 6

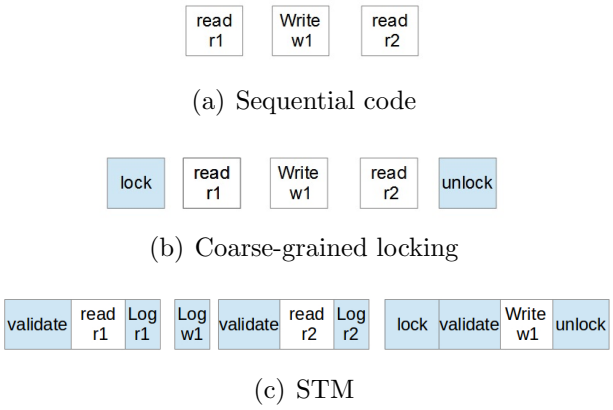
Remote Invalidation

In this chapter, we present *Remote Invalidation* (RInval), an STM algorithm which applies the same idea of RTC on invalidation-based STM algorithms. 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, like 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 validation and commit overheads. Then, we show how RInval significantly optimizes the transaction critical path.

6.1 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 6.1 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 6.1: Critical path of execution for: (a) sequential, (b) lock-based, and (c) STM-based code

In Figure 6.1(a), the sequential code contains only shared-memory reads and writes without any overhead. Coarse-grained locking, in Figure 6.1(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 [39]. 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 6.1(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 [39].

Locking. We already studied this overhead when we discussed RTC in Chapter 5, 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 [19]. However, the same idea can be applied to any other STM algorithm.

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.

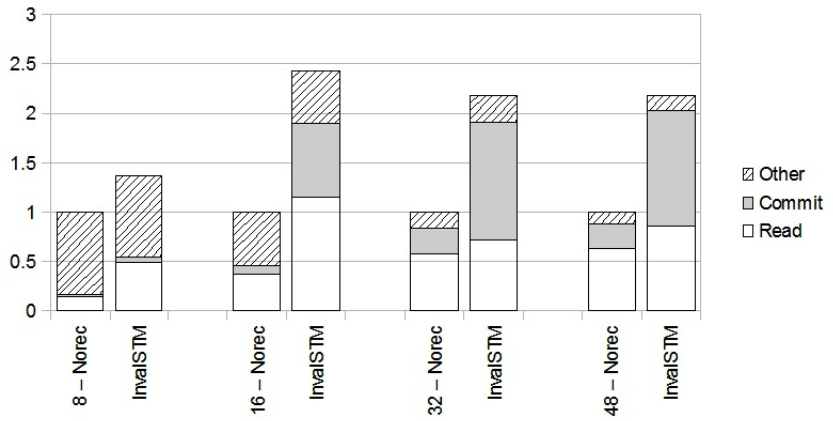


Figure 6.2: Percentage of validation, commit, and other (non-transactional) overheads on a red-black tree. The y-axis is the normalized (to NOrec) execution time

Figure 6.2 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 6.2 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.

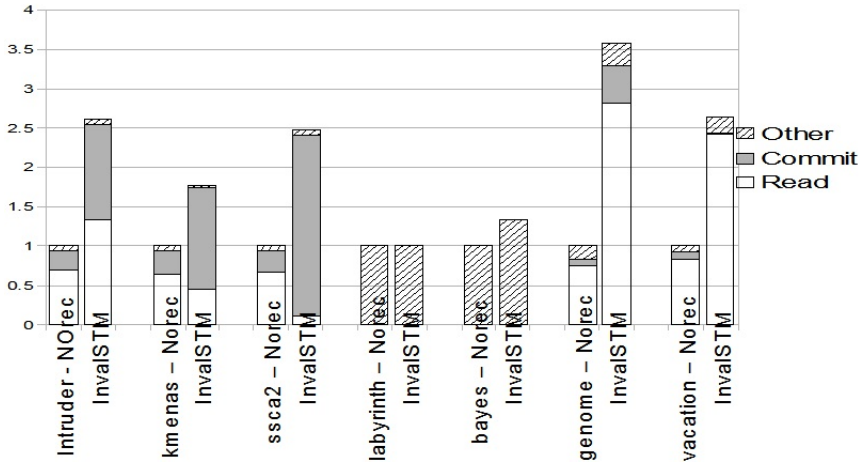


Figure 6.3: Percentage of validation, commit, and other (non-transactional) overheads on STAMP benchmark. The y-axis is the normalized (to NOrec) execution time

The same conclusion is given in the STAMP benchmark[2]. In Figure 6.3, 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

[2]We excluded yada applications of STAMP as it evidenced errors (segmentation faults) when we tested it on RSTM.

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 to minimize the effect of most of the previously mentioned overheads. Basically, we alleviate the effect of *i)* locking overhead, *ii)* the tradeoff between validation and commit overheads, and *iii)* 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.

6.2 Remote Invalidation

As described in Section 6.1, *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[3]. 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.

6.2.1 Version 1: Managing the locking overhead

RInval-V1 uses the same idea of RTC: commit routines are executed remotely to replace spin locks. Figure 6.4 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 6.5 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)*

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

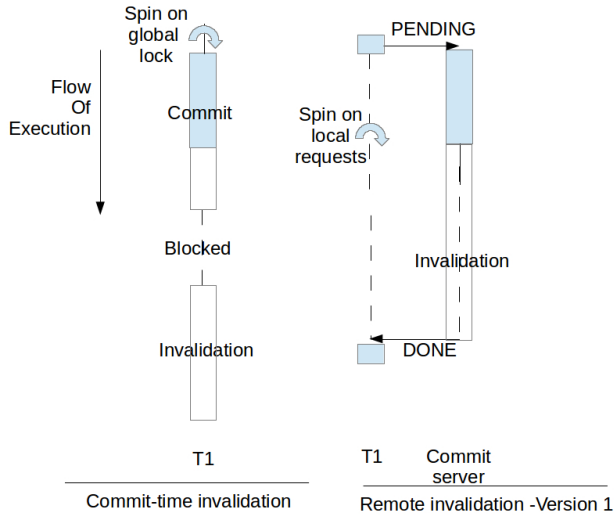


Figure 6.4: Flow of commit execution in both InvalSTM and RInval-V1

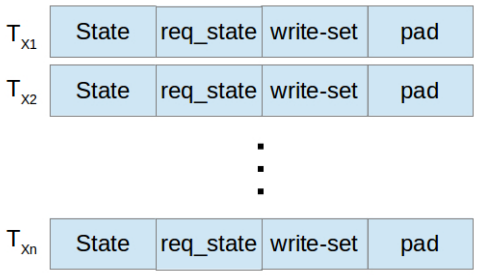


Figure 6.5: RInval's Cache-aligned requests array

more cores have to be dedicated for servers, *iii)* more CAS operations must be added to 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 12 shows the pseudo code of RInval-V1[4]. 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

[4]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).

Algorithm 12 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 ∉ (COMMITTED, ABORTED)
 9: end procedure

10: procedure Commit-Server Loop
11:     while true do
12:         for i ← 1, num_transactions do
13:             req ← 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
```

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 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.

6.2.2 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 6.6 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.

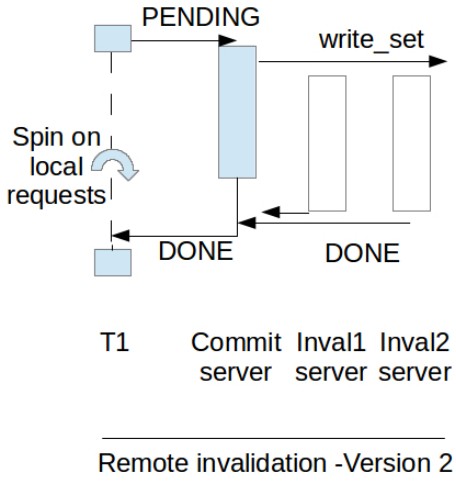


Figure 6.6: Flow of commit execution in RInval-V2

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 13 Remote Invalidation - Version 2

```
 1: procedure Commit-Server Loop
 2:     while true do
 3:         for i ← 1, num_transactions do
 4:             req ← requests_array[i]
 5:             if req.request_state = PENDING then
 6:                 for i ← 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 ← 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 13 shows RInval-V2's pseudo code. The client's commit procedure is exactly 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).

6.2.3 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 14 Remote Invalidation - Version 3

```
 1: procedure COMMIT-SERVER LOOP
 2:     if req.request_state = PENDING and req.inval_timestamp ≥ timestamp then
 3:         ...
 4:     ...
 5:     while timestamp > inval_timestamp + num_steps_ahead do
 6:         LOOP
 7:     ...
 8:     commit_bf[my_index++] ← 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 14 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.

6.2.4 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 6.1, 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).

6.2.5 Correctness and Features

RInval guarantees opacity in the same way other coarse-grained locking algorithms do, such as NOrec [19] and InvalSTM [27]. 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 [64], and easy integration with hardware transactions [60]. Hardware transactions need only synchronize with the commit-server, because it is the only thread that writes to shared memory.

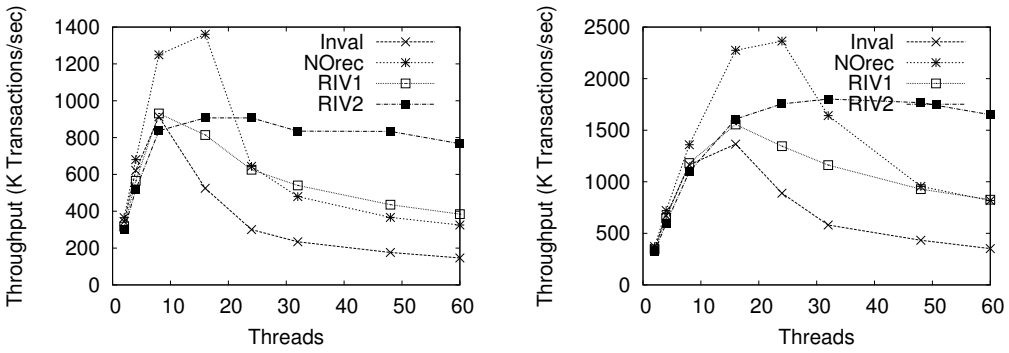
6.3 Evaluation

We implemented RInval in C/C++ (compiled with gcc 4.6) and ported to the RSTM framework [48] (compiled using default configurations) to be tested using its interface. Our ex-

periments 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 same strong properties as RInval, like minimal meta-data, easy integration with HTM, and privatization safety. We compared RInval with InvalSTM [27], the corresponding non-remote invalidation-based algorithm, and NOrec [19], 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)

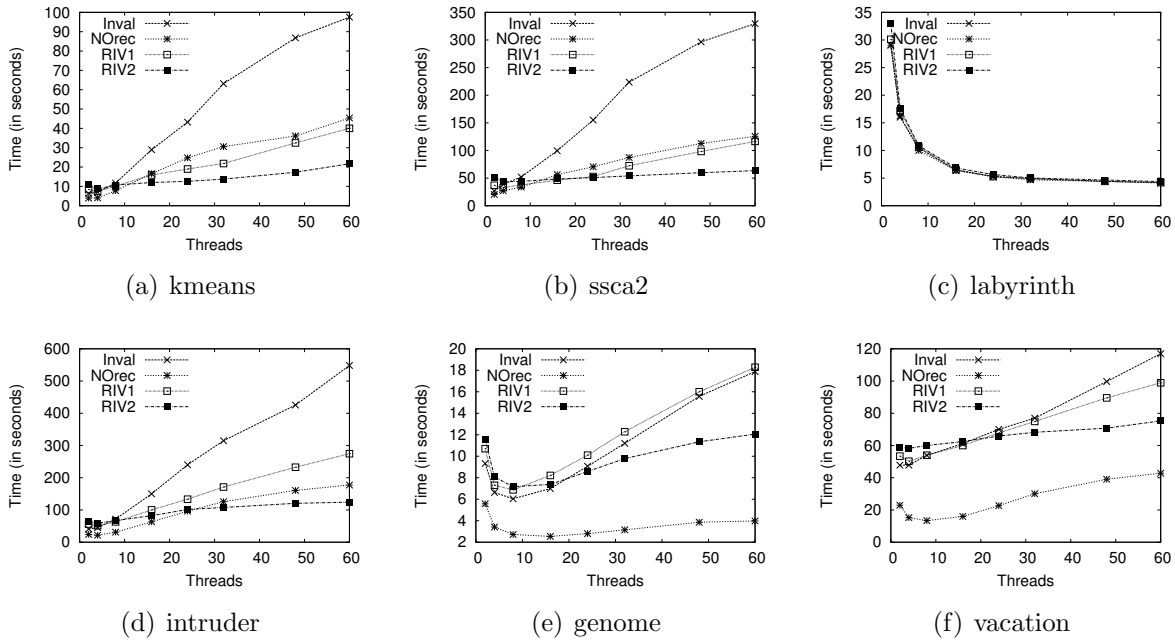
We show results in red-black tree micro-benchmark and the STAMP benchmark [53]. In these experiments, we show how RInval solves the problem of InvalSTM and becomes better than NOrec in most of the cases. All of the data points shown are averaged over 5 runs.



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

Figure 6.7: Throughput (K Transactions per second) on red-black tree with 64K elements

Red-Black Tree. Figure 6.7 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 6.8: 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 6.8 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 6.3. 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 6.3. 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 6.1, because their main overhead is non-transactional. For brevity, we did not show *bayes* as it behaves the same as *labyrinth*.

6.4 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.

Our work shows 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 7

Conclusions

In this dissertation proposal we made several contributions aimed at improving the performance of Transactional Memory. We analyzed the overheads that affect the performance of STM algorithms and classified them into three categories: locking mechanisms, transaction's critical path, and false conflicts. We tackled all these overheads in different ways. We presented OTB, a novel methodology for boosting concurrent data structures to be transactional. Then, we presented RTC a new validation-based STM algorithm which uses an efficient remote core locking mechanism instead of spin locking. We also presented RInval a corresponding invalidation-based STM algorithm which further optimizes transactions' critical path by isolating commit and invalidation routines in different server cores.

OTB showed promising results when compared to the original *pessimistic* boosting methodology. It performs closer to the optimized concurrent (non-transactional) data structures than the pessimistic boosting. It also performs up to an order of magnitude better than pure-STM data structures. In addition to the performance gains over pessimistic boosting, OTB allows more data-structure specific optimizations, and avoids the need to define inverse operations. We also showed how to integrate OTB with DEUCE Java framework. Our integration allows executing both memory-level and semantic-level reads/writes in the same transaction without losing the performance gains of OTB.

RTC outperforms other STM algorithms, by up to 4x, 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 2x faster than the corresponding validation-based STM algorithm (NOrec) and up to an order of magnitude faster than the corresponding invalidation-based STM algorithm.

7.1 Proposed Post-Prelim Work

7.1.1 Exploiting Intel's TSX extensions

Since the release of Intel's Haswell processor, many research proposals focus on finding the best fall-back path for the HTM transactions [13, 20, 50, 49]. Analyzing the results of the current research in literature, we conclude three main lessons. First, any alternative should keep the interference with the (fast) HTM path minimized. Fall-back algorithms that needs less instrumentation inside the common hardware *fast path* [13, 20, 50] are more promising than those which need more complicated hardware instrumentation [49]. Second, pure STM algorithms are still in the scene. In some cases, when HTM transactions will likely abort (because of capacity or non-supported operations), starting the transaction in STM from the beginning show the best performance. Third, HTM can be used in parts of the transaction rather than the whole transaction. This approach shows significant improvement either in general TM algorithms like RH-NOrec [50] or in concurrent data structures like in Hybrid-COP [6].

Based on our observations, we propose two major extensions to our work:

- Using RTC 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, 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, run transactions in software from the beginning, ...).
- Executing the commit phase of OTB in HTM transactions. Our analysis on the overhead of software-based spin locking, as well as recent similar analysis in literature [22], show that avoiding the costly CAS operations can result in a significant performance improvement. We expect similar improvement when OTB replaces these costly CAS operations with HTM blocks.

7.1.2 More OTB Data Structures

OTB is a methodology that can be applied to any lazy concurrent data structure. In Section 3.1, we introduced two basic rules for applying OTB. The first rule only requires that the concurrent algorithm has an (unmonitored) traversal phase, which is common to all lazy data structures. The second rule gives guidelines on how to make specific modifications to the underlying lazy data structure so that its correctness is preserved and its design is optimized for high performance. We propose extending OTB's library with balanced trees and maps.

Balanced Tree

Balanced binary search trees are data structures whose self-balancing guarantees logarithmic time complexity for `add`, `remove`, and `contains` operations. One of the main issues in balanced trees is the need for rotations to re-balance the tree. In AVL trees [3], rotations guarantee that the heights of the left and right subtrees of a node differ by no more than one. Red-Black trees [8] need less frequent re-balancing as they allow the longest length to be no more than twice the shortest length.

Although rotations complicate the design of any concurrent (or transactional) balanced tree, some solutions have been previously proposed [10, 17] for the design of efficient concurrent (lazy) balanced trees. Among these proposals, the concurrent tree of Bronson *et al.* [10] represents a good basic data structure for OTB. This tree has three main properties that fit OTB's rules. First, the tree is traversed using a hand-over-hand (time-based) validation mechanism, which means that, only a constant number of nodes are validated in each step. Second, 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 contradicts hand-over-hand validation [10].) Third, re-balancing is relaxed so that it uses local heights. Although other threads may concurrently modify these heights (resulting in a temporarily unbalanced tree), a sequence of localized operations on the tree eventually results in a strict balanced tree [10, 45].

Map

Map is a key-value store data structure. Each map entry has a unique key and an associated value that can be added, removed, modified, or queried. The main difference between map and set is the `update` operation, which modifies the value of a node with a certain key. However, adding this operation does not affect OTB's applicability. To implement the `update` operation, the map is traversed exactly like set. Then, if there is a node that matches the searched key, it is added to the semantic write-set to be locked and modified at commit; otherwise, the node is added in the same way as the set's `add` operation.

Although skip-lists and balanced trees can be used to implement maps, hash tables are more commonly used to implement maps. OTB hash tables can extend concurrent hash tables [39] using the same idea presented in Section 3.2.1. Hashing a key to its corresponding entry represents the (unmonitored) traversal phase of OTB. To resolve collisions, hash tables use either closed addressing or open addressing approaches. Hash tables with closed addressing seem more suitable for OTB than those with open addressing, because they use another data structure (e.g., skip-list or balanced tree) to store the keys with the same hash values. This way, OTB hash tables can easily be implemented as an extension of those data structures. Hash tables with open addressing are more tricky as they use probing to find a new entry in case of collisions. This probing contradicts OTB's unmonitored traversal and complicates

the design of the OTB map.

7.1.3 Enhancing RTC Contention Management

The worst case of RTC and RInval, when commit time is very small relative to the overall execution time, can be enhanced by involving the contention manager. Allowing strategies other than default strategy (in which the committing transaction always wins) may alleviate the effect of aborting the in-flight transactions. If a committing writing transaction conflicts with many in-flight transactions, it's better to abort (or pause) the committing transaction until the in-flight transactions successfully commit. InvalSTM [27] uses some of these strategies to improve the performance. Using the same ideas in RTC and RInval can result in better improvements because the decision will be globally made inside the servers.

7.1.4 Complete TM Framework

One of our goals in this dissertation is to introduce a complete TM framework which includes all of our enhancements. We use two different frameworks in our experimentation. The first framework is DEUCE, a Java STM framework which provides a simple programming interface without any additions to the JVM. The second fraework is RSTM, a corresponding C/C++ STM framework. Although there are some differences on how to write transactions, and how the underlying framework interpret transaction's code, both DEUCE and RSTM follow the same abstraction concepts of transactional memory. In this section we show how we plan to extend DEUCE framework to include all of our enhancements.

Figure 7.1 shows our proposal for modifying DEUCE framework. The light gray blocks are the blocks that we already implemented. The dark gray blocks are our proposed post-preliminary work.

To maintain the simple API of DEUCE, only few modifications on the first (application) layer are made. The programmer is provided with an extended API so that he can call OTB methods, and he can define the HTM blocks and their fall-back paths. However, the programmer remains not involved in the concurrency control at all. RTC and RInval are embedded in the third (TM algorithms) layer, and the proposed modified CM is attached to the second (runtime) layer. OTB data structures are embedded in the runtime layer, and they are communicating with the appropriate STM algorithms in the third layer, as we discussed in Chapter 4.

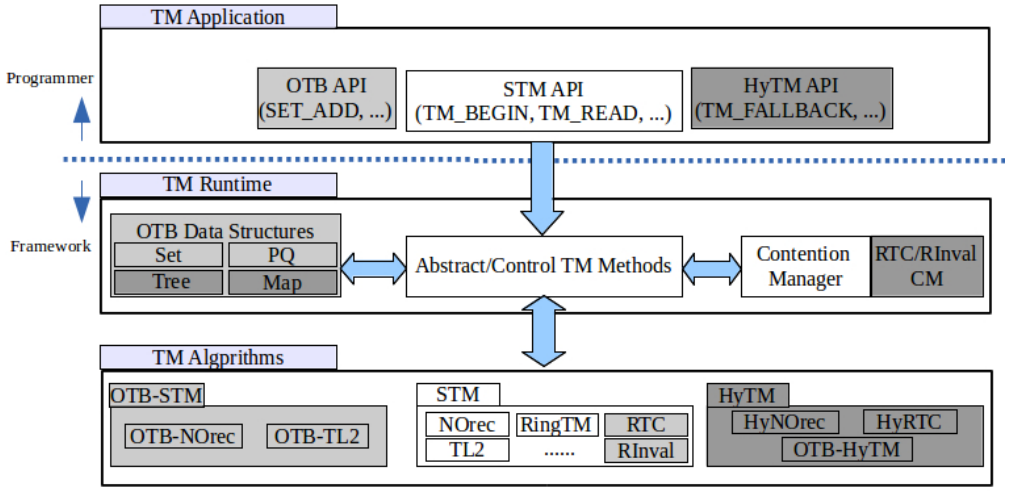


Figure 7.1: DEUCE STM framework with the proposed enhancements.

Bibliography

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

[2] Advanced Micro Devices, Inc. Advanced Synchronization Facility – Proposed Architectural Specification, 2.1 edition. Available `http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, 2009.

[3] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 145, pages 263–266, 1963.

[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] 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.

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

[7] 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.

[8] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *Micro, IEEE*, 29(2): 6–16, 2009.

[16] D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[22] 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.

[23] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.

[24] 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.

[25] 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.

[26] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[27] 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.

[28] 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.

[29] 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.

[30] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[31] T. Harris and S. Stipic. Abstract nested transactions. *TRANSACT 07: 2nd Workshop on Transactional Computing.*, August, 2007.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[39] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[40] 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.

[41] Intel Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[42] J. Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 2013.

[43] 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.

[44] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[45] K. S. Larsen. Avl trees with relaxed balance. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 888–893. IEEE, 1994.

[46] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, 2012.

[47] 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.

[48] 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.

[49] 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.

[50] A. Matveev and N. Shavit. Reduced hardware norec: An opaque obstruction-free and privatizing hytm. *TRANSACT 14: 9th Workshop on Transactional Computing.*, March, 2014.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] J. Moss and A. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[56] 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.

[57] 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.

[58] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.

[59] 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.

[60] 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.

[61] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

[62] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10 (2):99–116, 1997.

[63] 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.

[64] 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.

[65] 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.

[66] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional mutex locks. In *SIGPLAN Workshop on Transactional Computing*, 2009.

[67] TM Specication Drafting Group. Draft specification of transactional language constructs for c++, version 1.1, 2012.

[68] M. Tremblay. Transactional memory for a modern microprocessor. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 1–1, 2007. ISBN 978-1-59593-616-5.

[69] 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.