Abstract

A Study of Conflict Detection in Software Transactional Memory

Daniel Lupei
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2009

Transactional Memory (TM) has been proposed as a simpler parallel programming model compared to the traditional locking model. However, uptake from the programming community has been slow, primarily because performance issues of software-based TM strategies are not well understood.

In this thesis we conduct a systematic analysis of conflict scenarios that may emerge when enforcing correctness between conflicting transactions. We find that some combinations of conflict detection and resolution strategies perform better than others depending on the conflict patterns in the application. We validate our findings by implementing several concurrency control strategies, and by measuring their relative performance.

Based on these observations, we introduce partial rollbacks as a mechanism for effectively compensating the variability in the TM algorithm performance. We show that using this mechanism we can obtain close to the overall best performance for a range of conflict patterns in a synthetically generated workload and a realistic game application.
Contents

1 Introduction 1

2 Background 5
  2.1 Correctness Requirements ............................................. 5
  2.2 Concurrency Control ...................................................... 8
    2.2.1 Conflict Detection .................................................. 8
    2.2.2 Conflict Resolution ................................................ 9

3 Conflict Pattern Analysis 11
  3.1 Enforcing Validity Ranges ............................................. 12
  3.2 Enforcing Exclusiveness Ranges ...................................... 16

4 Uniform Conflict Detection based on Partial Rollbacks 21
  4.1 Scenarios Exhibiting Validity Ranges ............................... 22
  4.2 Scenarios Exhibiting Exclusiveness Ranges ......................... 24

5 libTM Design 26
  5.1 TM Algorithm Design .................................................... 26
    5.1.1 Timing of Conflict Detection ..................................... 27
    5.1.2 Conflict Resolution Strategy .................................... 29
  5.2 Rollback Mechanism ..................................................... 30
    5.2.1 Discussion ......................................................... 31
List of Figures

2.1 Possible correct schedules that enforce the validity range of $T_1$ with respect to $T_2$. ................................................. 6

2.2 Possible correct schedules that enforce the exclusiveness ranges between $T_1$ and $T_2$. ....................................................... 7

3.1 Execution parameters and optimum conflict detection/resolution for a RAW conflict, with $c_r > c_w$. ........................................ 13

3.2 Execution parameters and optimum conflict detection/resolution for a WAR conflict, with $c_r > c_w$. ........................................ 14

3.3 Summary of the optimum conflict detection policies under different execution parameters in scenarios where two transactions need to comply with a validity range. ........................................ 15

3.4 Execution parameters and optimum conflict detection/resolution for a WAR-WAR conflict pattern. ........................................ 17

3.5 Execution parameters and optimum conflict detection/resolution for a WAR-RAW conflict pattern. ........................................ 17

3.6 Execution parameters and optimum conflict detection/resolution for a RAW-WAR conflict pattern. ........................................ 18

3.7 Execution parameters and optimum conflict detection/resolution for a RAW-RAW conflict pattern. ........................................ 19
3.8 Summary of the optimum conflict detection policies under different conflict patterns where \textit{exclusiveness ranges} need to be enforced. \\

4.1 Through the use of checkpoints and partial rollbacks, the amount of work wasted in case of an abort is greatly reduced. \\

4.2 A lazy strategy augmented with support for \textit{partial rollbacks} achieves results similar or superior to those of an eager approach when enforcing \textit{validity ranges} between transactions. \\

4.3 A lazy strategy augmented with support for \textit{partial rollbacks} achieves results similar or superior to those of an eager approach when enforcing \textit{exclusiveness ranges} between transactions. \\

5.1 Type of conflict detection per conflict type for each conflict detection strategy. \\

5.2 Transactions executing under the pessimistic strategy. \\

5.3 Transactions executing under the read-optimistic strategy. Transaction T2 gets a chance to commit before the writing transaction triggers the invalidation. \\

6.1 Acquiring/releasing read/write access to a memory location based on its associated transactional lock. The transactional lock is represented by two bitmaps (as required by the read-optimistic policies) in order to individually illustrate each step of the procedure for obtaining write access. \\

6.2 Wait-for graph for deadlock detection. \\

6.3 All the possible configurations supported by our TM library in terms of conflict detection policies and rollback mechanisms. \\

6.4 The structure of the write-set that supports checkpointing and partial rollbacks.
7.1 The performance of different conflict detection and resolution policies under conflict patterns featuring \textit{validity ranges}. ........................................ 48
7.2 The performance of different conflict detection and resolution policies under conflict patterns featuring \textit{exclusiveness ranges}. ........................................ 49
7.3 The performance of different conflict detection and resolution policies in the presence of write-write conflicts. ........................................ 50
7.4 The performance of different conflict detection and resolution policies when 4 threads perform insert and remove operations on a hashtable. ........ 52
7.5 The performance of different conflict detection policies for the SynQuake game benchmark in a high contention scenario. ................. 54
7.6 Bookkeeping related overheads for write operations when using different rollback mechanisms. ......................................................... 55
7.7 Bookkeeping related overheads for read operations under different conflict resolution policies. ......................................................... 55
7.8 Locking related overheads for read and write operations under different conflict detection policies. ..................................................... 56
Chapter 1

Introduction

Transactional memory (TM) has recently emerged as a novel parallel programming paradigm for facilitating more efficient, programmer-friendly use of the plentiful parallelism available in hardware [11, 13].

The main idea is to simplify application programming in distributed and parallel environments through the use of transactions. TM allows transactions on different processors to manipulate shared in-memory data structures concurrently in a data-race-free manner. TM thus replaces traditional locking synchronization in parallel programs with a much simpler programming interface. Instead of explicit fine-grained locking of data items, the programmer specifies the beginning and the end of parallel regions with transaction delimiters. A runtime TM system (e.g., based on a software library linked with the program) automatically detects data races between concurrent transactions and ensures correct parallel execution for generic parallel programs. Any detected incorrect execution resulting from a data race causes one or more transactions to be rolled back and restarted. The run-time system automatically detects which memory regions are read and written by a transaction, and maintains the recoverability of data for the written ranges of memory.

In spite of the availability of several commercial and research prototypes supporting
TM in software, TM uptake from the programming community has been slow. One of the major reasons has been the fact that although a wide variety of TM algorithms and their implementations exist across the various research prototypes, from pessimistic lock-based TM algorithms to fully optimistic ones and a range of in-between hybrids, there is no systematic analysis and comparison of these implementations based on application conflict patterns. Consequently, no clear understanding exists with regard to what type of TM performs best for different types of application patterns.

Furthermore, previous literature offers contradictory accounts with regard to what kind of policy would provide the best overall performance. Spear et al. [26] argue that a fully optimistic approach as adopted in TLII [4] or JudoSTM [19] should be preferred since it significantly reduces the probability of livelock occurring between transactions by holding transactional locks for only a brief period of time. On the other hand Felber et al. justify opting for a pessimistic policy in their implementation of TinySTM [8] with a range of empirical results supporting the idea that detecting conflicts early and aborting “doomed” transactions as soon as possible leads to an important reduction in the amount of wasted work. They also point out that optimistic designs require more expensive mechanisms for handling read-after-write “hazards” during the execution of a transaction. Finally, in an attempt to extract the benefits of both approaches, hybrid policies have been proposed by Spear et al. [27] and Dragojević et. al [6], but unfortunately their viability has only been underpinned by empirical evaluations.

In this thesis, we perform a comprehensive study of all the conflict scenarios that may emerge between two transactions. The examined scenarios are derived from correctness criteria, as proposed by Guerraoui and Kapalka [12] or by Scott [24], that any STM has to enforce between conflicting transactions. We observe that the optimum conflict detection policy varies from one conflict pattern to another, while also being highly dependent on the specific execution parameters (such as expected commit time) of one transaction with respect to the other. The optimistic strategies with lazy conflict detection tend
to perform best when conflicts can be easily sidestepped or if the nature of the conflict doesn’t preclude transactions executing concurrently. In contrast, pessimistic approaches are most advantageous in scenarios where conflicts cannot be avoided, e.g., deadlock situations, and thus it is crucial to provide an ordering between transactions as soon as possible such that the amount of work wasted when solving them is minimized.

The resulting variability in what strategy performs best makes it impossible for any of the algorithms previously explored in the literature to provide an optimum detection of conflicts. Consequently, we introduce a mechanism that provides support for *partial rollbacks*, in order to effectively mitigate inherent conflict detection policy pit-falls and provide stable performance close to the best approach in all cases.

By allowing transactions to perform *partial rollbacks*, we can successfully alleviate the risks of wasted work associated with optimistic protocols. With support for *partial rollbacks*, an optimistic read transaction reverts back only to a checkpoint close to the initial optimistic read upon being aborted by a conflicting writer. In optimistic protocols, the risk of wasted work exists mainly for the case of abort resolution of read-write conflicts, whereas write-write conflicts can be solved by providing support for multiple writers. Therefore, our *partial rollback* support compensates for the risk of an optimistic read, making a fully optimistic approach appealing in all cases.

We introduce a software transactional memory library, called libTM, which implements a broad range of TM algorithms, which vary in their approach to conflict detection (from fully-pessimistic to fully-optimistic), conflict resolution mechanism (wait-for versus abort) and recoverability mechanism.

We evaluate the performance of eight different TM protocol versions, and our risk compensation mechanism, providing support for *partial rollbacks*, on synthetically generated workloads as well as on an existing game benchmark, called SynQuake. SynQuake is a realistic game benchmark that emulates the processing performed by the Quake game-engine as well as its data structures, while allowing us to easily create different game
We find that the performance of different TM strategies is sensitive to the application’s conflict pattern. For example, pessimistic approaches with eager conflict detection perform best in scenarios where read-write conflicts have a high probability of forming a cycle and write-write conflicts are infrequent. On the other hand, fully optimistic protocols with lazy detection of conflicts and support for multiple writers perform best for write-write conflicts.

While the best TM algorithm varies with the conflict pattern, our results show that a fully-optimistic strategy with partial rollback support consistently delivers the best performance across all the conflict patterns tested.
Chapter 2

Background

In this chapter, we first look at the correctness criteria that must be enforced by transactions competing for the same shared objects. We will then present the concurrency control mechanisms used by STMs in order to provide the previously described correctness guarantees.

2.1 Correctness Requirements

Maintaining consistency and ensuring safety are the two major concerns in any STM. Consequently, the STM must comply with correctness criteria, such as the ones discussed by Guerraoui [12] or Scott [24], when handling conflicts between transactions.

Strict serializability [20] is the most widely provided property by transactional memory prototypes. Informally, it requires that operations performed by all committed transactions return the same values as if the execution of these transactions has been serialized in a particular order.

However, strict serializability doesn’t provide safety guarantees in execution environments that aren’t sandboxed, where a transaction executing based on an inconsistent state of memory may enter a behavior from which it can no longer recover (e.g. divide by zero exception). Consequently, additional restrictions have to be formulated addressing
“in-flight” transactions as well. Towards this end, Guerraoui and Kapalka have proposed opacity [12] as a safety property requiring that not only committed transactions, but also live ones always operate on a consistent view of the memory.

These properties can be enforced, as shown in [24], by making sure that all objects read during a transaction remain valid from the time of the access until the transaction either commits or aborts. Therefore, concurrent transactions aren’t allowed to apply any updates during that period of time. We will be referring to this interval as the validity range of the transaction reading an object with respect to the transaction writing it. If the intersection of the read set of one transaction with the write set of another contains more than one object, the validity range is defined by the first occurring read access from the intersection set.

Figure 2.1 outlines the possible ordering of two transactions, $T_1$ and $T_2$, whose read set and write set intersect, with $T_1$ reading object $A$ and $T_2$ updating it. In the first case, $T_1$ executes concurrently with $T_2$ during it’s validity range and only at the end we have the update of $A$ scheduled after the commit of $T_1$. In the second case, the entire
validity range of \( T_1 \) is serialized after transaction \( T_2 \)’s commit of the updated value of \( A \). Under both schedules the validity range between the point where \( T_1 \) reads \( A \) and the time when it finishes is not allowed to overlap with any operation that might mutate object \( A \), in our example: the commit of \( T_2 \).

Two transactions may both have validity ranges with respect to one another, as a result of one’s read-set intersecting the other’s write-set and vice-versa. These will act as exclusiveness ranges during which any overlap between their executions will cause one of the transactions to become invalid. Consequently, in order to produce correct schedules, the STM has to serialize one of the exclusiveness ranges after the other, as illustrated in Figure 2.2.

With respect to updates, transactions must make them visible and apply them to shared objects only when they commit. This is true regardless of when exactly the write accesses took place during the course of the transaction.
Chapter 2. Background

2.2 Concurrency Control

A typical STM operates on shared objects via read or write accesses. Two transactions are said to conflict if both of them try to access the same shared object $A$ and at least one of them is trying to perform a write. Depending on the order of the accesses, three types of conflict may arise:

- **read-after-write (RAW)**, where one transaction reads $A$ after the other wrote it,
- **write-after-read (WAR)**, where the write of one transaction follows the other’s read,
- **write-after-write (WAW)**, where both transactions write to $A$ successively.

STMs enforce the correctness requirements discussed in section 2.1 by using a series of concurrency control mechanisms which take decisions based only on the observed conflicts between transactions. First, as part of the conflict detection strategy, they have to decide when exactly should conflicts between transactions be detected. Once a conflict is detected, it is the conflict resolution policy that decides which of the involved transactions is the winning one and will be allowed to continue, and which of them is the victim being forced to either abort or wait for the winning transaction to finish.

2.2.1 Conflict Detection

The conflict detection strategy can choose to acknowledge conflicts either eagerly, at encounter-time, resulting in a pessimistic policy, or lazily, at commit-time, in the case of an optimistic policy. This decision has a very significant impact on performance, since detecting conflicts early can save a lot of work in “doomed” transactions, while postponing it until commit-time can allow for more concurrency in cases where conflicts can be ordered such that none of the involved transactions has to be invalidated.

The conflict detection policy can vary based on the type of conflict being detected, resulting in mixed approaches. The most common mixed conflict detection strategy
performs early detection of WAW conflicts, while WAR or RAW conflicts are being handled lazily.

### 2.2.2 Conflict Resolution

The conflict resolution policy, also referred to as contention management, acts like a scheduler for conflicting transactions, its decisions being critical in avoiding pathological conditions like livelock or starvation.

Several strategies have been proposed in the literature varying in the amount of additional context information that they require [11, 10, 16, 26]. The simpler approaches include the Aggressive policy, that always allows the transaction detecting the conflict to continue by aborting the other one, and the Polite policy which handles a conflict by having the transaction discovering it wait for the other one to finish. When applied only to WAR conflicts we refer to these policies as Wait-For Readers and Abort Readers, respectively. The more complex solutions include policies like Greedy [11] or Polka [16] that take into account information regarding the starting order of the transactions or the amount of work performed up to the current time.

Conflict detection and conflict resolution policies appear to work independent of one another, with the first one deciding when to detect a conflict and the second one specifying how to solve it. However, Dragojević et al. [6] pointed out that this is not the case. They showed that on testing combinations of two conflict detection policies, eager and lazy, and two conflict resolution policies Greedy and Polka, on the STMBench7 benchmark, the best and worst results were both obtained with the same eager conflict detection policy. Consequently, the best concurrency control strategy cannot be established by dividing the process in two steps, as previously attempted [16, 27], with the first one determining the best approach to conflict detection and the second one exploring the best solution for conflict resolution.
In this thesis, we will mainly be focusing on the conflict detection aspect of concurrency control and we will analyze its impact on all possible conflict patterns that may emerge between two transactions when enforcing the correctness requirements outlined in Section 2.1. Since the best conflict detection policy varies across conflict scenarios, we will propose a mechanism for providing a unified approach to conflict detection that performs near-optimally under any conflict pattern. As an important consequence, this will also allow future research on concurrency control to evaluate different conflict resolution strategies in isolation and to avoid the explosion in design space caused by the interdependency between conflict detection and conflict resolution policies.
Chapter 3

Conflict Pattern Analysis

In this chapter, we identify all relevant conflict scenarios where validity and exclusiveness ranges have to be enforced, by varying the conflict pattern between transactions, as well as the timing of their accesses. We then establish the optimum conflict detection policy for each such scenario. We will point out that none of the traditional approaches to conflict detection, eager or lazy, achieves optimum results under all conflict patterns and execution scenarios.

Since the scope of a decision taken by the conflict detection policy includes a single conflict between two transactions, our analysis will only feature scenarios involving two transactions.

We say that a conflict detection policy makes an optimum decision if it leads to the minimum amount of wasted work or time. In order to properly evaluate the outcome of each considered scenario, we will consider both possible schedules that a conflict resolution strategy may enforce, and we will determine the optimum approach to conflict detection in both cases.
3.1 Enforcing Validity Ranges

Since an STM has no apriori knowledge regarding the read-set or write-set of a transaction, it has no precise way of determining validity ranges beforehand. As a result, its conflict detection and resolution policies make decisions based on individual conflicts encountered during the execution. In our analysis, we consider both types of conflict, RAW and WAR, that can induce a validity range between two transactions, by having one transaction reading a shared object $A$, while the other one updates it.

**Observation.** A WAW conflict between two transactions attempting to write to the same object does not impose a validity range, since transactions are only required to make their writes public when they commit. As a result, these types of conflict can be easily handled with a lazy approach that allows both transactions to modify the object at their finishing point in the order in which they commit.

**Notations.** For the rest of this chapter, when referring to scenarios featuring a single validity range we will be using the following notations: $T_r/T_w$ represents the transaction performing a read/write on a shared object $A$, $s_r/s_w$ represents the starting time of transaction $T_r/T_w$, $t_r/t_w$ represents the time of the read/write access on object $A$ and $c_r/c_w$ represents the committing point of transaction $T_r/T_w$ if allowed to run in isolation.

**RAW conflict.** Figure 3.1a outlines the case of a RAW conflict where we have transaction $T_r$ reading object $A$ after transaction $T_w$ modified it ($t_w < t_r$). Two situations may arise: i) transaction $T_r$ finishes before $T_w$ ($c_r < c_w$) or ii) $T_r$ commits after transaction $T_w$ ($c_r > c_w$). In the first situation, we can sidestep the conflict and have both transactions executing concurrently and eventually committing by delaying conflict detection until commit time. If we allow $T_r$ to read the old value of $A$, then by the time $T_w$ wants to commit and update $A$, $T_r$ would have already finished and released it. This strategy ensures correctness without any wasted time or work.

In the second situation, where $c_r > c_w$, the conflict can no longer be avoided, and
Figure 3.1: Execution parameters and optimum conflict detection/resolution for a RAW conflict, with \( c_r > c_w \).

the STM needs to enforce an explicit ordering between transactions. A \( T_w \) before \( T_r \) schedule can be best enacted by having \( T_r \) wait at the time of the read access until \( T_w \) commits and updates the value of \( A \). This approach, corresponding to an eager conflict detection policy, leads to \( c_w - t_r \) time wasted while waiting, as outlined in Figure 3.1b. The alternative of a lazy policy would have \( T_r \) deferring the conflict and reading the old value of \( A \). However, later on, \( T_w \) will have to invalidate \( T_r \) when the new value of \( A \) is committed such that the \( T_w \leftarrow T_r \) ordering is put into effect. This results in \( c_w - s_r \) wasted time due to the abort, which is clearly worse than the \( c_w - t_r \) provided by the eager alternative (as \( s_r \parallel t_r \)).

The opposite ordering, transaction \( T_r \) before \( T_w \), can be implemented optimally using a lazy policy, by having \( T_w \) wait at commit time until transaction \( T_r \) completes and relinquishes the use of object \( A \). This second ordering, illustrated in Figure 3.1c, results in \( c_r - c_w \) wasted time while waiting.

Depending on the defining parameters of the transactions: \( t_r \), \( c_r \) and \( c_w \), one of the two orderings (\( T_w \leftarrow T_r \) or \( T_r \leftarrow T_w \)) with corresponding penalties of \( c_w - t_r \) and \( c_r - c_w \) will yield better results. Since these schedules are obtained through different conflict
Figure 3.2: Execution parameters and optimum conflict detection/resolution for a WAR conflict, with $c_r > c_w$.

detection policies (eager vs. lazy) it is not possible to have one approach that will work best in both scenarios.

**WAR conflict.** The case of a WAR conflict has transaction $T_w$ updating object $A$ after $T_r$ has read it ($t_r < t_w$). Similar to the RAW conflict, a distinction arises between i) the situation where $T_r$ intends to finish first ($c_r < c_w$) and ii) the reverse case, where $T_w$ would commit before $T_r$ ($c_w < c_r$). As previously discussed, in the first scenario, a lazy policy offers the optimal outcome since it effectively avoids the conflict by allowing $T_r$ to use the old value of $A$.

The second situation, presented in Figure 3.2a, requires that the conflict be explicitly addressed. The STM again has a choice of ordering $T_r$ before $T_w$ or vice-versa. If $T_r$ is to commit first, $T_w$ has to delay its update of object $A$ until $T_r$ finishes and no longer requires its use. This can best be achieved by a lazy conflict detection policy, since it leads to the least amount of time wasted waiting: $c_r - c_w$ (Figure 3.2b).

For the reverse ordering, $T_w$ before $T_r$, the commit of $T_w$ will invalidate the value of $A$ read by $T_r$, causing it to abort. The sooner $T_r$ aborts, the less time and work is wasted as a result. That is why an eager approach which has $T_w$ invalidating $T_r$ at the
Figure 3.3: Summary of the optimum conflict detection policies under different execution parameters in scenarios where two transactions need to comply with a *validity range*.

Since the actual costs of each schedule are dependent on the timing of the operations performed by each transaction and since each schedule is the result of a different conflict detection policy, we again face the dilemma of not having an universal approach that will consistently provide optimum results.

The discussion from this section is summarized in Figure 3.3. In all but two cases, a lazy conflict detection policy will lead to the minimum amount of wasted work when enforcing a *validity range* between two transactions.
3.2 Enforcing Exclusiveness Ranges

When two transactions both have validity ranges with respect to one another, they act as exclusiveness ranges, during which the execution of the two transactions needs to be serialized. Since STM’s conflict detection and resolution policies work at the level of conflicts, we identified all the possible conflict patterns that can lead to exclusiveness ranges between two transactions. We did that by looking at all possible interleavings occurring between two transactions, $T_1$ and $T_2$, with $T_1$’s read-set intersecting $T_2$’s write-set and vice-versa. More precisely, we had $T_1$ reading an object $A$ and writing to another object $B$, while $T_2$ read $B$ and updated $A$. All possible interleavings resulted in one of the following conflict patterns: WAR-WAR, WAR-RAW, RAW-WAR and RAW-RAW. In this section, we investigate the optimum conflict detection policy when enforcing exclusiveness ranges for each of these conflict patterns.

**Notations.** For the rest of this chapter, when referring to scenarios with exclusiveness ranges we will be using the following notations: $T_x$ represents transaction $x$, $s_x$ and $c_x$ represent the starting and finishing time of transaction $x$ if it were to be run in isolation, $\text{RAW}_x$ and $\text{WAR}_x$ represent conflicts detected at transaction $x$. Also $t_{\text{acc}}(X)$ will denote the time when access $\text{acc}$ is performed on object $X$, where $\text{acc}$ can be either $\text{rd}$ or $\text{wr}$ and $X$ can be either $A$ or $B$.

**WAR-WAR conflict pattern.** Figure 3.4a presents two transactions whose interleaved accesses generate a WAR-WAR conflict pattern. In this scenario, the STM has to provide a scheduling that enforces exclusiveness ranges: $c_1 - t_{\text{rd}}(A)$ and $c_2 - t_{\text{rd}}(B)$. Since by the time any conflict is detected there is already an overlap between the two exclusiveness ranges, the required serialization can only be achieved by aborting one of the transactions. In order to minimize the amount of wasted work, we need to employ an eager conflict detection policy that would trigger the abort as soon as the transaction allowed to commit first discovers the WAR conflict. This approach would result in
max\( (t_{wr(B)} - s_2, c_1 - t_{rd(B)}) \) or \( \max(t_{wr(A)} - s_1, c_2 - t_{rd(A)}) \) wasted work depending on whether \( T_1 \) gets scheduled before \( T_2 \) or vice-versa (Figure 3.4b and Figure 3.4c).

**WAR-RAW conflict pattern.** An example of two transactions instantiating a
Chapter 3. Conflict Pattern Analysis

Figure 3.6: Execution parameters and optimum conflict detection/resolution for a RAW-WAR conflict pattern.

WAR-RAW conflict pattern is depicted in Figure 3.5a. Correct execution can be achieved in this scenario, by either delaying $T_2$’s read of $B$ until $T_1$ commits, or by aborting transaction $T_1$ and rescheduling its read of $A$ after $T_2$ finishes.

The first approach, of ordering $T_1$ before $T_2$, see Figure 3.5b, results in $c_1 - t_{rd}(B)$ time wasted waiting. When applying it, STM will have to eagerly resolve the RAW conflict emerged from $T_2$’s read of $B$, by forcing it to wait for $T_1$’s commit of the new value of $B$. Also, the WAR conflict, triggered by $T_2$’s update of $A$, should be handled lazily, since for this schedule, we want to maintain the objects read by $T_1$ valid until it eventually commits.

The alternative ordering, $T_2$ before $T_1$, can be optimally put into effect by aborting $T_1$ as early as possible, since the value of $A$ it read will become outdated once $T_2$ completes. In our case, the STM should eagerly resolve the WAR conflict by invalidating $T_1$, causing it to waste $\max(t_{wr}(A) - s_1, c_2 - t_{rd}(A))$ time due to the abort (Figure 3.5c).

RAW-WAR conflict pattern. Although the order of conflicts is reversed from the previously discussed scenario, the policies required to efficiently ensure correctness
for the RAW-WAR conflict pattern are very similar. An example of two transactions exhibiting this type of conflict pattern is provided in Figure 3.6a. As before, the $T_2$ after $T_1$ ordering should be enforced by detecting the RAW conflict early and having $T_2$ wait until $T_1$ commits the updated value of $B$ (Figure 3.6b). For the reverse schedule, $T_2$ before $T_1$, $T_2$ would have to postpone the detection of the RAW conflict until commit-time, while eagerly aborting $T_1$ upon the detection of the WAR conflict (Figure 3.6c).

**RAW-RAW conflict pattern.** Figure 3.7a outlines transactions whose *exclusiveness ranges* are a result of RAW conflicts. In this scenarios, correct scheduling can be most advantageously obtained by having the transaction ordered last wait when attempting its read access, until the other transaction commits and releases the object at the root of the conflict. Consequently, the most appropriate conflict detection and resolution strategy would have an eager approach for the RAW conflict encountered by the transaction ordered last, while deferring the RAW conflict encountered by the transaction ordered first. Such a strategy will yield $c_1 - t_{rd}(B)$ or $c_2 - t_{rd}(A)$ in wasted time while waiting, depending whether $T_1$ or $T_2$ is allowed to commit first (see Figure 3.7b, 3.7c).
<table>
<thead>
<tr>
<th>Conflict Pattern</th>
<th>Schedule</th>
<th>1&lt;sup&gt;st&lt;/sup&gt; conflict</th>
<th>2&lt;sup&gt;nd&lt;/sup&gt; conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Conflict Detection</td>
<td>Conflict Resolution</td>
</tr>
<tr>
<td>WAR&lt;sub&gt;1&lt;/sub&gt;-WAR&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a) $T_1 \leftarrow T_2$</td>
<td>eager</td>
<td>abort</td>
</tr>
<tr>
<td></td>
<td>b) $T_2 \leftarrow T_1$</td>
<td>lazy</td>
<td>-</td>
</tr>
<tr>
<td>WAR&lt;sub&gt;2&lt;/sub&gt;-RAW&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a) $T_1 \leftarrow T_2$</td>
<td>lazy</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>b) $T_2 \leftarrow T_1$</td>
<td>eager</td>
<td>abort</td>
</tr>
<tr>
<td>RAW&lt;sub&gt;2&lt;/sub&gt;-WAR&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a) $T_1 \leftarrow T_2$</td>
<td>eager</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>b) $T_2 \leftarrow T_1$</td>
<td>lazy</td>
<td>-</td>
</tr>
<tr>
<td>RAW&lt;sub&gt;1&lt;/sub&gt;-RAW&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a) $T_1 \leftarrow T_2$</td>
<td>lazy</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>b) $T_2 \leftarrow T_1$</td>
<td>eager</td>
<td>wait</td>
</tr>
</tbody>
</table>

Figure 3.8: Summary of the optimum conflict detection policies under different conflict patterns where exclusiveness ranges need to be enforced.

Figure 3.8 summarizes the analysis presented in this section and confirms the results from the previous section: that no single conflict detection and resolution strategy can provide optimum results under all conflict patterns and execution parameters. However, for the scenarios discussed in this section, we can see that the eager conflict detection policy dominates, since an early detection of conflicts is crucial in providing an efficient serialization of exclusiveness ranges.
Chapter 4

Uniform Conflict Detection based on Partial Rollbacks

We propose the use of partial rollbacks combined with a lazy approach to conflict detection in order to provide a unique policy that will make near-optimal decisions under any conflict pattern that may emerge between two transactions. In particular, we will show that the outcome for all the cases where an eager strategy is preferable, can be matched or outperformed by adding support for partial rollbacks to a policy that detects conflicts lazily.

Checkpointing and partial rollbacks can be used in transactional systems in order to limit the amount of work wasted when conflicts are being solved by aborting one of the transactions. This usually requires that transactions save their state periodically during their execution by taking checkpoints (see Figure 4.1). In the event of an abort due to a conflicting write on a shared location, the invalidated reading transaction would roll back only to the checkpoint taken just before the initial read access of that location, as opposed to the starting point of the transaction. The example presented in Figure 4.1 illustrates the reduction in the amount of wasted work resulting from the use of partial rollbacks when a WAR conflict is settled by aborting the reading transaction.
Figure 4.1: Through the use of checkpoints and partial rollbacks, the amount of work wasted in case of an abort is greatly reduced.

In the following, we will revisit some of the scenarios exhibiting validity or exclusiveness ranges that we analyzed in the previous sections. We will focus only on the cases where we concluded that an early detection of conflicts was most advantageous. We will show how the ability to partially rollback a transaction cancels the deficiencies of a lazy approach, allowing it to provide results similar or superior to those of an eager policy.

### 4.1 Scenarios Exhibiting Validity Ranges

For scenarios with transactions featuring validity ranges we identified two situations where an eager approach to conflict detection is required for optimum results. In the first one we have two transactions, $T_r$ and $T_w$, involved in a RAW conflict with $T_r$ reading the shared object $A$, while $T_w$ attempts to write it. If the expected commit time of $T_r$ exceeds that of $T_w$ ($c_w < c_r$) and we choose to order the writing transaction first ($T_w \leftarrow T_r$), then having $T_r$ wait at the time of the access for $T_w$ to commit the
Figure 4.2: A lazy strategy augmented with support for partial rollbacks achieves results similar or superior to those of an eager approach when enforcing validity ranges between transactions.

new value of $A$ leads to the minimum amount of wasted time: $c_w - t_r$. By contrast, a lazy strategy would allow $T_r$ to optimistically read $A$ and delay the detection of the conflict until $T_w$ commits. However, at that point $T_r$ will get invalidated resulting in an increased amount of wasted work: $c_w - s_r$. If the lazy strategy is to be provided with support for partial rollbacks, it would allow transaction $T_r$ to only rollback up to the point of the read access. This would consequently lead to only $c_w - t_r$ in wasted work, which is the exact amount of time wasted by the eager policy. The resulting schedule is presented in Figure 4.2a.

The second scenario where an eager approach to enforcing validity ranges outperforms a lazy one has two transactions, $T_r$ and $T_w$, conflicting in a WAR manner while accessing object $A$. As before, if the expected commit time of $T_r$ exceeds that of $T_w$ ($c_w < c_r$) and we choose to order the writing transaction first ($T_w \leftarrow T_r$), then aborting $T_r$ as soon as possible, that is when $T_w$ performs its write access, will yield the minimum amount of
work wasted: \( \max( t_w - s_r, c_w - t_r) \). Delaying the abort up to the commit of \( T_w \), as a lazy policy would do, will only increase the amount of wasted work to \( c_w - s_r \). However, a lazy policy with partial rollback support would have \( T_r \) only rollback up to the point where it read \( A \), as illustrated in Figure 4.2b. This will result in only \( c_w - t_r \) in wasted work, making it appear as if \( T_r \) clairvoyantly delayed its read until \( T_w \) committed a new value of object \( A \). More importantly, the outcome of this strategy represents only the lower bound on the cost incurred by the eager approach \( (c_w - t_r \leq \max( t_w - s_r, c_w - t_r)) \).

4.2 Scenarios Exhibiting Exclusiveness Ranges

In section 3.2 we analyzed 4 conflict patterns: WAR-WAR, WAR-RAW, RAW-WAR, RAW-RAW, where the STM needs to enforce exclusiveness ranges in order to guarantee correctness. In all of them, this can be optimally achieved by an early detection of conflicts. Depending on the types of conflicts that compose the pattern, it is either the transaction that is allowed to commit first that eagerly aborts the other one in case of WAR conflicts, or it is the transaction ordered last that delays its read access in case of RAW conflicts. For instance, in the pattern WAR-RAW depicted in Figure 3.5a, the \( T_1 \leftarrow T_2 \) ordering will be enforced by having \( T_2 \) wait before reading object \( B \), until \( T_1 \) finishes and commits an updated value. For the reverse ordering, \( T_2 \) before \( T_1 \), we have \( T_2 \) resolve the WAR conflict by aborting \( T_1 \) at the time of its write access of \( B \). In both situations, the key idea behind minimizing wasted time is the ability to perform conflict detection as soon as possible, which is something that doesn’t happen when a lazy policy is used. We propose adding support for partial rollbacks as a way of overcoming this deficiency in an attempt to create a conflict detection strategy that behaves optimally under any circumstances.

For the conflict pattern discussed above, the use of partial rollbacks makes it possible
Figure 4.3: A lazy strategy augmented with support for partial rollbacks achieves results similar or superior to those of an eager approach when enforcing exclusiveness ranges between transactions.

for a lazy approach to avoid the unfortunate consequence of the late detection of conflicts, which is the complete abort of the transaction ordered last. By contrast, the ordering of $T_1$ before $T_2$ can be established by a lazy strategy at the same cost as an eager one: $c_1 - t_{rd}(B)$; if, when being invalidated by the commit of $T_1$, $T_2$ has the option of rolling back only up to the point where it read $B$. Similarly, for the $T_2$ before $T_1$ ordering, having support for partial rollbacks allows $T_1$ to roll back only up to the read of object $A$. As a result, when the lazy policy finally detects the WAR conflict at the commit of $T_2$ and has to abort $T_1$, we end up wasting only $c_2 - t_{rd}(A)$, which is also the lower bound on the time wasted by an eager approach. Both scenarios are illustrated in Figure 4.3.

The same line of reasoning can also be applied to the rest of conflict patterns analyzed: WAR-WAR, RAW-WAR and RAW-RAW, leading to similar conclusions.
Chapter 5

libTM Design

In the following, we describe the TM algorithm implemented by our libTM library with the different policies it provides in terms of conflict detection and resolution. We will also discuss the different rollback mechanisms supported by our library for ensuring recoverability and their impact on overall performance.

5.1 TM Algorithm Design

We build a family of TM algorithm versions by varying the protocol design in two main dimensions. First, we vary the timing of conflict detection between: eager i.e., upon access and lazy i.e., at commit time. Second, we vary the conflict resolution strategy between: waiting for the conflicting transaction to commit versus aborting the conflicting transaction. In addition we vary the recovery mechanism between maintaining an undo log versus write buffering of modifications.

We select and investigate the algorithm versions that represent meaningful combinations of the above parameter settings. All of our protocols are variations of the classic two-phase locking concurrency control algorithm, resulting in a blocking implementation of transactional memory semantics. We prefer to build on the two-phase locking algorithm rather than on timestamp-ordering concurrency control, since two-phase locking
Chapter 5. libTM Design

offers higher flexibility with respect to conflict resolution. The timestamp-ordering approach always resolves conflicts in favor of the transaction that started first, whereas the two-phase locking approach poses no constraints on ordering.

The higher flexibility of two-phase locking thus enables us to freely explore the design space.

In the next sections, we describe our strategies for conflict detection and conflict resolution in all our TM algorithm versions. In accordance with the two-phase locking approach, our conflict detection and resolution policies base their decisions on read and write transactional locks acquired by transactions at different stages in their execution. In section 5.1.1 we focus on exploring the timing of conflict detection i.e., the eager versus lazy design decision, while in section 5.1.2 we discuss the conflict resolution strategy. Finally, we describe our recovery mechanisms in all possible protocol variants in section 5.2.

5.1.1 Timing of Conflict Detection

Our TM library implements several protocols that vary the timing of conflict detection for different types of conflicts between: **eager**, where conflicts between transactions are detected at access time, and **lazy**, where conflicts are detected at commit time.

Given the resulting design space, we select to implement the four strategies below, which are most commonly encountered in STM designs due to their efficient implementation under the two phase locking algorithm.

- fully pessimistic: read and write locks are acquired upon access,

- partially read-optimistic: write locks are acquired upon access, readers wait to gain access if a writer lock exists, otherwise readers proceed optimistically without locks.

- read-optimistic: write locks are acquired upon access, readers always proceed optimistically without locks.
Table 5.1 shows the resulting conflict detection configurations for all our strategies, for all types of conflict, i.e. read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW).

The **fully pessimistic** strategy enforces transactional read/write locks from the moment of the initial access up to the point of commit. For upgrading from a read to a write lock, the write lock is acquired first (preventing any new readers from obtaining the read lock) and then the writer resolves the conflicts with all other existing readers. Readers can gain access to an object only if the write lock is not held. Deadlocks are detected and resolved by using a wait-for graph.

In the **partially read-optimistic** strategy, transactions mark their writes within the transactional write lock upon access. This allows for early detection of WAW and RAW conflicts, with reading transactions having to wait for access if a writer lock exists. A writer delays conflict resolution with any of the pre-existing readers (the case of WAR conflicts) until commit-time.

In the **read-optimistic** strategy, transactions record their writes at the time of the access, but will eagerly detect only WAW conflicts. In contrast with the previous policy,
reading transactions are always allowed to proceed optimistically. The transaction performing the write will resolve all conflicts with readers at commit time (lazy detection of WAR and RAW conflicts).

In the fully-optimistic strategy both read and write accesses are performed optimistically, without locks. This allows multiple readers and multiple writers to access a location concurrently. At commit time, each writer obtains exclusive locks for all locations in its write set and resolves any existing conflicts with other transactions.

In general, the optimistic strategies trade the level of allowed concurrency on one hand for an increased amount of wasted work in case of aborted executions on the other hand.

### 5.1.2 Conflict Resolution Strategy

For all conflict detection strategies summarized in table 5.1, once a conflict is detected, the conflict can be resolved by either i) waiting for the conflicting transaction to commit or ii) aborting the conflicting transaction.

We study the trade-offs between these two conflict resolution strategies only for WAR conflicts because it is only for these types of conflict that we have a schedule (the reading transaction after the one performing the write) that can be enforced only by aborting the reading transaction (since the value it read will become invalid as a result of the writing transaction finishing first). For all other types of conflict, RAW and WAW, any of the possible orderings can be enacted by having the victim transaction wait after the transaction allowed to commit first. For instance, in the case of RAW conflicts, scheduling the writer first can be enforced by asking the reader to wait at the time of the access until the writer commits the updated object, while the reverse ordering can be achieved by allowing the reader to continue and forcing the writer to wait for the reader to finish before committing its update.

Consequently, in our library, RAW and WAW conflicts are always resolved by waiting
for the conflicting transaction to commit, and we vary the conflict resolution strategy only for WAR conflicts. Depending on the decision taken by the writer with respect to the conflicting readers, we call the two conflict resolution strategies: `waitfor_readers` and `abort_readers`. One may perform better than the other depending on the remaining work in the reader transaction(s) at the time of conflict (the waiting cost), versus the work already performed by the reader(s) (the abort cost).

While more complex conflict resolution strategies exist, their analysis is not the topic of our work. By contrast, we focus mainly on exploring the impact of conflict detection policies on the overall performance of STM systems and our conflict resolution strategies are designed simply to allow all optimum schedules of conflicting transactions as identified in sections 3.1 and 3.2.

## 5.2 Rollback Mechanism

Transactional memory updates become final only when a transaction commits. Consequently, a TM library needs to implement some mechanism for ensuring that uncommitted writes are not final. To rollback the modifications performed by a failed transaction, we can choose between maintaining an undo log or keeping uncommitted updates private in a write-buffer. Since the conflict detection strategy imposes some restrictions with respect to the type of rollback mechanism allowed and since none of these variants constitutes a very clear choice, we implemented both of them in our library in order to perform a detailed analysis of their benefits and drawbacks.

When using undo-logging, the original value of the accessed location is saved in a private buffer and restored in case of an abort. With write-buffering, all the writes are performed in a private buffer and the shared data is updated only in case of a commit. To avoid read-after-write hazards, the write-buffer needs to be searchable so that a thread is able to locate and correctly read data that it has previously written in the current
transaction. For performance, searchability can be implemented using a hashtable. By comparison, undo-logging requires only sequential access to its private buffer, making it less expensive to implement.

The rollback strategy presents another trade-off related to the number of times data is being copied between the shared memory and the private buffers. In the case of a successful transaction, undo-logging and write-buffering both require one copy; undo-logging performs it at access time, while write-buffering does it at commit time. However, for each unsuccessful transaction, undo-logging performs an additional two copies, one at access time and another when rolling back, while in the case of write-buffering the private buffers are simply discarded. This gives an advantage to write-buffering in scenarios with high abort rates.

With respect to conflict detection, undo-logging can only be used under the pessimistic strategy. That is because the pessimistic strategy is the only one where the writer is granted exclusive rights over the accessed location at encounter time.

5.2.1 Discussion

With the pessimistic and the read-optimistic strategies one can choose between resolving conflicts with readers at encounter time or at commit time. The first variant, shown in Figure 5.2, has the advantage that it offers writers exclusive rights over the accessed location so that undo-logging can be used, which is less expensive to implement. On the other hand, using write-buffering and allowing readers to continue until the writer commits may provide them with the opportunity to finish their transactions before that happens and in the end may lead to a lower amount of wasted time. Such a scenario is presented in Figure 5.3, where transaction T2 manages to avoid invalidation when conflict resolution is performed by the writing transaction at commit time. However, in this case, since we have multiple readers and one writer having access to the shared data concurrently, the writer needs to use a write-buffer as its rollback mechanism, which is a
Figure 5.2: Transactions executing under the pessimistic strategy.

Figure 5.3: Transactions executing under the read-optimistic strategy. Transaction T2 gets a chance to commit before the writing transaction triggers the invalidation.

In section 6.5 we will discuss an optimization that alleviates the cost of using a write buffer for strategies that resolve read-write conflicts at commit time and write-write conflicts eagerly.
Chapter 6

libTM - API and Implementation

Details

In this chapter, we will first present the interface exposed to the programmer for transactionalizing parallel applications. We will then describe the meta data we used to support our library’s conflict detection and resolution strategies, as well as the mechanisms for supporting partial rollback of transactions.

6.1 libTM Library Interface

Applications synchronizing parallel tasks using our libTM library need to have their transactions delineated with begin_transaction and commit_transaction statements.

Furthermore, a distinction has to be drawn between shared and private per-thread data accessed during the course of a transaction. This is necessary since in the case of private data libTM has to provide only rollback support for recovery from abort events. However, for shared data, additional tracking of accesses performed by concurrent threads is needed in order to ensure consistency. For this purpose, libTM requires that transactional shared and private variables be declared using two distinct meta-types tm_shared and tm_private, respectively. As an example, a shared variable, int x in the
original program, needs to be declared as \texttt{tm\_shared<int>x}.

Since the entire state of the execution needs to be restored in case of an abort, libTM maintains recovery data for both \texttt{tm\_shared} and \texttt{tm\_private} variables updated during a transaction. However, conflict detection and resolution is performed only when accessing \texttt{tm\_shared} variables. Consequently, the meta data (transactional lock) associated with a shared variable in support of the conflict detection and resolution policies is also encapsulated as a member of the \texttt{tm\_shared} meta-type.

The definition of each of these meta-types in our library is a C++ template using the original type of the variable as a parameter, (e.g. \texttt{tm\_shared<original type>}). In order to allow run-time monitoring of accesses on shared and private transactional variables, their implementations overload the conversion operator for tracking reads and the assignment operators for tracking writes.

Although we perform access tracking at word level granularity as a result of our approach based on operator overloading, libTM provides support for variable transactional locking granularity by allowing the programmer to map multiple \texttt{tm\_shared} variables to a single transactional lock.

### 6.2 Transactional Locks

Conflict detection and resolution policies as well as other aspects of the TM algorithm (i.e., state management) are dependent on tracking meta-data, also called transactional locks, maintained as part of the \texttt{tm\_shared} meta-type. In libTM, transactional locks are represented by bitmaps, where each bit is associated with one of the active threads. The last (exclusiveness) bit is reserved for indicating whether the lock is held in exclusive mode or not, by a writing thread.

When performing a read, a thread will be allowed to proceed and register its access by setting its corresponding bit in the transactional lock, only if the exclusiveness bit is not
Figure 6.1: Acquiring/releasing read/write access to a memory location based on its associated transactional lock. The transactional lock is represented by two bitmaps (as required by the read-optimistic policies) in order to individually illustrate each step of the procedure for obtaining write access.

Set. This allows us to vary the type of detection for RAW conflicts, between eager or lazy, by having the writing thread set the exclusiveness bit either at access or commit time. Furthermore if a writing thread gains ownership of a lock at access time, it can choose between solving WAR conflicts with pre-existent readers at the same time or delaying their resolution until commit-time.

Setting the exclusiveness bit, solving conflicts with other readers along with establishing exclusiveness relative to other writers are the three procedures that a transaction needs to perform when writing a location in order to detect all possible types of conflict: RAW, WAR and WAW. The difference between all the conflict detection policies explored comes from the timing of the application of these procedures during the execution of the transaction.
For the fully-pessimistic strategy, which detects all conflicts eagerly, setting the exclusiveness bit and solving conflicts with pre-existent readers will both take place when performing the write access. Additionally, the bit corresponding to the writing thread’s id will also be set in order to ensure exclusiveness with respect to other threads attempting to write the same object.

When allowing WAR conflicts to be detected lazily, as the partially read-optimistic policy does, the resolution of conflicts with pre-existent readers is postponed until commit time, while still setting the exclusiveness bit at access time. Since pre-existent readers continue to be tracked on the transactional lock’s bitmap past the moment of the access, writer threads have to ensure exclusiveness among themselves by setting their corresponding bit on an additional bitmap, which will function as a write lock.

The read-optimistic policy relaxes conflict detection even further, by performing late detection of both WAR and RAW conflicts. As a result, all readers are allowed to proceed concurrently with the writer, which will set the exclusiveness bit and solve conflicts with readers only at commit time. Early detection of WAW conflicts is enforced similar to the partially read-optimistic strategy.

Finally, the actions performed by the fully optimistic approach resemble those of the fully-pessimistic policy with the main distinction that in the former everything takes place at commit-time. Also, there is no need for the additional write lock anymore since at that point the writer gains full ownership of the transactional lock, and can enforce the late detection of WAW conflicts based on its bitmap.

The process of acquiring/releasing read/write access to a memory location based on its associated transactional lock is presented in Figure 6.1. In the figure transactional locks are represented by two bitmaps, in order to individually illustrate each step of the procedure for acquiring write access. However, as mentioned before, the additional bitmap (write lock) is necessary only in the conflict detection policies that resolve conflicts with pre-existent readers at a later time than establishing exclusiveness relative to other
writers (i.e., the read optimistic strategies).

Due to the size of a word, we currently only accommodate up to 31 threads executing transactions simultaneously. However, this is not a hard limit, since on 64 bit platforms there is support for executing atomic operations on up to 128 bits.

The procedure for recording a read access is similar across all conflict detection strategies and it consists of atomically setting the bit corresponding to the reading thread’s id on the lock word. Read access to a location can be delayed if a writer has already acquired that lock exclusively, or it can be revoked if the abort readers conflict resolution is in use and another thread is trying to gain exclusive access for writing. For the last case we say that the lock word also fulfills the function of a visible readers set, used in the process of invalidating all transactions that have read the location which is about to be written.

For the pessimistic strategy, a transaction upgrades to write access by first atomically setting the exclusive bit, thereby ensuring that no new threads can acquire read access. Then, depending on the conflict resolution policy used, it can either wait for all existing readers to finish their transactions and release their locks or it can abort them and clear their positions on the lock word. After invalidating the readers, the only bits still set are the exclusive bit and the bit corresponding to the writer’s thread id.

For the read-optimistic strategies, a second lock word called the write lock is used for managing write accesses, since the first word will be used as a visible readers set, and register all reading threads allowed to proceed concurrently until the writing transaction commits.

When acquiring write access, the corresponding bit of the current thread’s id is set on the write lock. Then, the writer will block any new reader from gaining access by setting the exclusive bit of the read lock. Depending on whether the RAW conflicts are detected eagerly or lazily, this step takes place either at access time or at commit time. Finally, before committing, the writer resolves any outstanding conflicts with the readers.
6.3 Deadlock Detection

If a conflict between two transactions is solved by having one of them wait for the other we need to make sure that a deadlock never occurs. Consequently, we prevent such conditions by ensuring that no cycles ever develop in a graph tracking the \textit{wait-for} relationship between all threads blocked in their attempt to gain read or write access to a shared location.
The wait-for graph contains a node for each thread. An edge exists from a thread $P_i$ to another thread $P_j$ if there is a transactional lock $L$ held by $P_j$ and requested by $P_i$. There is a deadlock if and only if a cycle has formed in this graph.

The wait-for graph is implemented as an array, where each element of the array corresponds to a specific thread or node in the graph. If a thread is waiting to gain access to a location, the address of the transactional lock protecting that location will be registered in its element in the array. Otherwise, that entry will be left void. Since the current owners of a lock are marked on its bitmap, we can easily determine the outgoing edges of a thread in the wait-for graph by inspecting the transactional lock it is currently trying to acquire as recorded in its corresponding entry in the array.

We test for cycles by starting from the node associated with the current thread and recursively exploring outgoing edges while keeping track of the visited nodes. The recursion stops either when reaching a node with no outgoing edges (a void entry in the array) or when visiting a node for a second time, signaling the presence of a cycle. When a cycle is discovered, the thread that detected it aborts itself, hence effectively breaking the deadlock. The process of detecting cycles is illustrated in Figure 6.2.

6.4 Invalidation Strategy

In order to provide consistency in the context of the abort-readers conflict resolution strategy, it is necessary for a transaction to become aware of the fact that the shared state on which it relies has been modified.

Our TM library addresses this problem by using an invalidation strategy that relies on visible-readers. Specifically, every reader records its access of a memory location in the transactional lock associated with that memory location, which will also fulfill the function of a visible-readers set. A transaction updating a location will set the abort status of all transactions reading that location at the time when WAR conflicts are being
solved. To avoid any inconsistent executions, a transaction checks its abort status at every operation on shared state.

The alternative solution would be to have the reader check whether the version of any previously read object (location) has changed, before allowing it to read a new object. However, this strategy incurs high overheads because of its $O(n^2)$ complexity, with $n$ being the number of reads. It has been previously shown that even if this check is performed only when handling an exception and only every $m$ iterations inside a loop, it still generates high overheads [23].

### 6.5 Transaction State Management

Checking whether the current transaction already has read or write access for a location is done either by inspecting the bitmaps of the transactional lock associated with that location, or by searching the write-set of the transaction. The second situation occurs when writing to a location under the fully-optimistic conflict detection strategy, which acquires write locks only at commit-time.

When reading a location, if write-buffering is used, the first step is to check whether the current transaction has written that location in the past. If this is the case, the address of the updated value from the write-buffer is returned instead. Otherwise, if this is the first time reading the location we try to obtain read access and if successful we record it in the read-set.

If `abort_readers` is used as the conflict resolution policy, another step is required: a temporary copy of the value read needs to be made, followed by an abort status check. This is necessary in order to avoid scenarios where an update of the location between acquiring read access and reading the value goes undetected, in which case the transaction will end up working on inconsistent data. However, this additional step raises the cost of performing a read under the `abort_readers` conflict resolution policy.
On every write we check whether the transaction already holds the write access and if this is not the case, we acquire it and record it in its write-set. For the fully-optimistic strategy the acquiring step is postponed until the pre-commit phase.

During the pre-commit phase, if the conflict detection strategy is fully-optimistic, all the write locks corresponding to locations in the write set are acquired and conflicts with the transactions reading them are solved. If the conflict detection strategy is read-optimistic then only the conflict resolution step takes place in this phase.

At commit time, if a write-buffer is used, we copy all updated values from the transaction’s write-set to shared memory. By contrast, when undo-logging is used, the log can simply be discarded. In case of an abort, the situation gets reversed, and we have to restore the original values back to shared memory when undo-logging is used, but we can simply discard the buffers in the case of write buffering. Finally, in both cases, commit or abort, all locks corresponding to the write and read sets are released.

The write-buffer needs to be searchable since during a transaction a read following a write to the same location has to return the latest value. Specifically, the value placed in the write-buffer needs to be returned, rather than the value from the shared location. However, this search can be avoided by adding a pointer to the meta data associated with each shared location. Consequently, the writer can save the address of the aforementioned data in this pointer, and reference it directly. On any subsequent read or write, a thread accesses the appropriate location in its write-buffer by following this pointer. With this optimization, we eliminate the main disadvantage of lazy detection of read-write conflicts, i.e., the need for a searchable write-buffer.

Unfortunately, this optimization cannot be applied to the fully-optimistic strategy requiring write-buffering, since this strategy allows multiple concurrent writers. We cannot afford to add a pointer for each of the writers to the meta data of a shared location, because the memory overhead makes this prohibitive. All viable configurations in terms of conflict detection and rollback mechanism are presented in Figure 6.3.
Every set used for bookkeeping is implemented as a sequential buffer [15], which is a linked list of buffers that can grow or shrink dynamically at a cost amortized by the fact that memory allocations/deallocations are done infrequently and in large chunks. When we need a searchable data structure, which is the case for the write-buffer with a fully-optimistic conflict detection strategy, we fit an open addressing hash table with a Bloom-filter [3] on top of the sequential buffer. Effectively this strategy allows us to have low dynamic memory allocation overheads while still being able to perform efficient look-up operations.

Figure 6.4: The structure of the write-set that supports checkpointing and partial rollbacks.
### 6.6 Checkpointing for Partial Rollbacks

Since the read and write sets are implemented with sequential buffers, all locations read and written will be maintained in the order they were initially accessed. This facilitates checkpointing as it allows us to record the state of a transaction with respect to the locations accessed by simply saving the position it reached in its read and write set. Additionally, if a location is re-written after a checkpoint, a new record needs to be created for it in the transaction’s write-buffer since the old value might have to be restored in the event of a partial rollback up to that checkpoint. Finally, in order to appropriately resume execution in case of a partial rollback, the current execution context (instruction pointer, stack pointer, etc.) is also stored as part of the checkpoint.

When aborted by a conflicting write transaction, the transaction that performed the optimistic read searches its read-set for the first location that has been invalidated by the write transaction. Once found, based on its position in the read-set, it will effectively determine the checkpoint that the transaction needs to rollback to as being the last checkpoint taken before reading the invalidated location.

During rollback, the transaction discards from its write-set all records corresponding to updates performed after the checkpoint, and releases all locks other than those found in the read and write-sets prior to the checkpoint. This step is simplified by the sequential nature of these sets. Finally the checkpointed execution context is restored and the execution resumed, correspondingly.

The cost of creating a checkpoint has two components: a fixed and relatively small one incurred when saving the execution context as well as the positions reached in the read and write sets; and a dynamic one, resulting from the need to keep extra copies of the objects written across checkpoints. However, for all practical reasons this second component remains fairly small since, in general, applications have an overwhelming majority of read accesses.
6.6.1 Write-set Support for Partial Rollbacks

Figure 6.4 shows the structure of a write-set that has support for partial rollbacks. On the first level we have a Bloom-filter, which can efficiently determine whether a location is part of the write-set or not, but allowing false positives. An open-addressing hashtable constitutes the next level, providing fast search into the write-buffer situated on the third level. The write-buffer is implemented by a sequential buffer and holds an entry for each location that was written during the transaction, in the order that it was accessed. Finally, on the last level we have the values-set maintaining the values of the locations registered into the write-set, with each entry in the write-buffer pointing towards its current value in the values-set. If a location is modified after a checkpoint is taken, a new value is added to the values-set, becoming the working copy for that location. Each entry added to the values-set is tagged with the id of the current checkpoint, and it will be joined in a linked list with the previous entry corresponding to the same memory location. This makes it very easy, in the event of a partial rollback, to determine for each location in the write-set which of the values should become the current one.

The connections that form in the write-set between the open-addressing hashtable, the write-buffer and the values-set, in the case of a transaction with two checkpoints, are exemplified in Figure 6.4. Location A is written before the first checkpoint, and then updated after checkpoint 1 and 2. As a result, the values-set will have three entries corresponding to location A, linked together in the order they were created, with the write-buffer entry pointing towards the most recent one. Similar bookkeeping can be observed for location B, modified before and after checkpoint 2 and location C updated only after checkpoint 2.

If the transaction is partially rolled back up to checkpoint 2, all locations added to the write-set before that point, will have their write-buffer entries redirected towards their values in the values-set created before the checkpoint was taken. These values can be identified by simply following the links that join all the entries in the values-set.
corresponding to a specific location until we reach the first one with a tag equal or smaller than 2, which is the id of the checkpoint we’re reverting to. In our example, write-buffer entries for location A and B will be redirected towards values A2 and B2, respectively. The rollback is completed with the disposal from the write-buffer and the values-set of all entries added after checkpoint 2 was recorded.
Chapter 7

Experimental Results

We evaluated the performance of all the conflict detection and resolution strategies explored on a series of conflict scenarios as well as a benchmark operating on a hashtable. We show that in all test cases the fully-optimistic policy with support for partial rollbacks provides the best performance. Finally, we also compare the overheads incurred by the mechanisms used for implementing the conflict detection strategies.

7.1 Experimental Setup

For our experiments we used an Intel Quad Core 2 at 2.66 GHz with four 64KB L1 caches, two 4MB L2 caches, and 2GB of RAM. We compiled the benchmarks with the O3 optimization flag and averaged results over three runs of each experiment.

7.2 Conflict Detection Policies under Different Conflict Patterns

We showcase the differences between conflict detection policies, as outlined by the analysis in Chapter 3, by looking at several conflict scenarios exhibiting either validity or
exclusiveness ranges. In order to generate the relevant conflict patterns as accurately as possible, we restrict our first experiments to only two threads, continuously executing transactions in a while loop, and limiting their accesses of shared data to only two locations. Additionally, we model the computation performed by transactions by inserting 1K "nop" instructions in between these accesses and between them and the start and the finish point of the transaction, respectively. By varying the pattern of access to the shared locations we can obtain specific conflict patterns where either a validity range or exclusiveness ranges need to be enforced between the transactions executed by each thread.

In each test we measure the throughput over a period of 30 seconds, for each of the four conflict detection strategies, combined with all our conflict resolution policies (waitfor_readers or abort_readers with and without support for partial rollbacks).

We first examine a test case with one of the threads executing transactions that read shared locations A and B, and the other executing transactions that write them. This results in a conflict scenario where the STM needs to enforce the validity range of the first thread reading location A with respect to the update applied at commit-time by the second thread. Since it is possible for both transactions to execute concurrently and eventually commit successfully, the conflict detection policies that allow optimistic detection of read-write conflicts provide the best performance, as can be observed from the results presented in Figure 7.1.

In the next three scenarios, presented in Figure 7.2, each thread executes a mix of read and write accesses, resulting in exclusiveness ranges with respect to one another. For test case 7.2a, with one thread reading location A followed by writing location B, and the other thread reading B followed by writing A, the pessimistic policy performs better than any of the alternatives. Since in this scenario deadlocks can easily occur, it is not always the case that both transactions can finish successfully. As a result, detecting conflicts eagerly helps limit the amount of wasted work. The downside of a conflict
Figure 7.1: The performance of different conflict detection and resolution policies under conflict patterns featuring *validity ranges*.

resolution policy that waits for readers, as opposed to aborting them as soon as possible, is also exemplified. In this case, waiting for the reader only makes things worse, since most of the times the abort is unavoidable.

The performance of conflict detection policies under WAR-RAW or RAW-RAW conflict patterns, is presented in Figures 7.2b and 7.2c, respectively. In both cases we see that allowing pre-existing readers to continue until the writer commits, but blocking new readers from gaining access, as the partially read-optimistic policy does, can indeed lead to the situation where both threads commit successfully, and therefore exploit the entire potential for parallelism. Additionally, the predilection for livelock of the fully pessimistic strategy, with readers being aborted as a result of a conflict, is also exposed, as it achieves very low performance in this scenario.
The use of partial rollbacks proves to be extremely beneficial for the optimistic strategies in all the test cases exhibiting exclusiveness ranges. With partial rollbacks, the usual price of being optimistic when reading a location, the risk of a full abort when the writer commits, is now replaced with that of returning to the point in the transaction
where the invalidated location was first accessed. This is in fact very similar to the price of being pessimistic and waiting there for the writer to commit, in the first place, as mentioned in Chapter 4.

Finally, we examine scenarios where both threads execute write-only transactions. Since the fully optimistic policy is the only one that detects WAW conflicts lazily and allows multiple writers to execute concurrently it is the only one that fully exploits the potential for parallelism of this scenario. This can also be observed from the results presented in Figure 7.3.

Overall, we can see that no conflict detection policy outperforms others in all scenarios, and each of them is better suited for a specific application conflict pattern.
However, when support for partial rollbacks is provided, the results show that the fully-optimistic strategy consistently delivers the best results.

### 7.3 Micro-benchmark Results

We also evaluated the performance of the conflict detection policies on a benchmark with 4 threads executing transactions consisting of insert and remove operations on a hashtable. In our tests we varied the number of operations per transaction as well as the size of the hashtable in terms of the number of buckets. We also fixed the load factor of the hashtable to 1000 elements, and when support for partial rollbacks was enabled, checkpoints were taken every 100 elements accessed.

From the results presented in Figure 7.4 we notice that the performance achieved when the `waitfor_readers` policy is used for resolving conflicts is overall lower than that delivered by the `abort_readers` policy. Consequently, we focus our discussion on the comparison between the results obtained by the `abort_readers` policy with or without support for partial rollbacks.

Across all sizes of the hashtable, we observe that at low numbers of operations per transaction, all policies behave similarly as the level of contention is relatively low. As we increase the number of operations per transaction, and without support for partial rollbacks, we see that the partially read-optimistic policy outperforms the others when the size of the hashtable is small (see Figure 7.4a). However, for larger hashtables the probability of a livelock occurring rises, and as a result the performance of the partially read-optimistic policy drops below that of the fully-optimistic and read-optimistic policies (see Figure 7.4c). Additionally, we notice that this policy provides relatively stable performance across all hashtable sizes. In contrast, when support for partial rollbacks is provided, the fully-optimistic policy matches the throughput obtained by the partially read-optimistic policy with small hashtables and outperforms it for larger hashtables.
Figure 7.4: The performance of different conflict detection and resolution policies when 4 threads perform insert and remove operations on a hashtable.

Hence, the fully-optimistic strategy becomes the best approach regardless of the workload parameters.
7.4 SynQuake Results

In this section we present results obtained by the conflict detection policies explored on a game benchmark called SynQuake. SynQuake is primarily based on the popular open-source Quake 3 game, but it is a full-fledged multiplayer game in its own right. SynQuake extracts representative features of many first person shooter games, or those of strategy games involving a mix of short-range and long-range interactions.

SynQuake models three types of game entities: players, resources (represented by apples) and walls. Each game entity is defined by its position on the game map and by a set of attributes specific to its type. For example, besides its position on the game map, a player is described by its life or health level, its speed and direction. The Quake 3 area node tree is used as a standard spatial data structure facilitating storing and retrieval of the location and attributes of game objects on the game map. Players are mutable game entities that can have both their position and attributes modified as a result of game interactions. For example, an attack decreases a player’s life, while consuming a resource increases it. Resources are partly mutable e.g., apples can have their attributes affected by game play, but not their position, while walls are immutable entities. As a result, each of these game objects requires different levels of synchronization in SynQuake, from full synchronization protection for players to no protection for walls. To simulate areas of high interest in the game, and the associated pattern of players flocking to a particular area of the map, we have added quests which attract players towards that area with a high probability. These correspond to standard areas attracting players existing in Quake 3, and also in strategy games, such as a camp site, weaponry location, and health areas.

Figure 7.5 presents, for each conflict detection policy, the time it took to execute 500 simulation cycles for 120 players executing a mix of move and eat operations. In order to provide a high level of contention most of the resources are placed around the center of the map where a quest is also activated. The results obtained when the \texttt{waitfor\_readers} conflict resolution policy is used are omitted as they either match those obtained by the
Figure 7.5: The performance of different conflict detection policies for the SynQuake game benchmark in a high contention scenario.

The abort-readers conflict resolution policy performs best for the pessimistic conflict detection policies (fully-pessimistic and partially read-optimistic), while registering a significant drop in performance for the more optimistic policies (read-optimistic and fully-optimistic). However, when support for partial rollbacks is provided we see that the results of the fully-optimistic policy become very close to those previously achieved only by the pessimistic policies. This confirms the fact that support for partial rollbacks can indeed allow the fully-optimistic policy to recover from its sub-optimal decisions while only incurring a small penalty related to recording checkpoints and performing partial rollbacks.

### 7.5 Overhead Analysis

Since the architecture of our library allows us to change the conflict detection strategy and the rollback mechanism, we also evaluated their relative performance. All the possible combinations are summarized in Figure 6.3. Out of 12 possible combinations only 8 can
Figure 7.6: Bookkeeping related overheads for write operations when using different rollback mechanisms.

Figure 7.7: Bookkeeping related overheads for read operations under different conflict resolution policies.

be used because some of the conflict detection strategies impose certain restrictions on the type of rollback mechanism that can be employed.

These series of experiments target absolute execution times of each strategy when
reading a location or writing to it during a successful or failed transaction. The experiment consists of a sequence of random reads or writes executed in a series of transactions. A total of 1M transactions are executed, where each individual transaction executes 1K reads or writes. Commit versus abort performance of writes is measured by forcing all transactions to commit or abort respectively.

To eliminate any variability caused by conflicting transactions we ran this experiment with only one thread. We split the execution time of each strategy in two: the time spent in transactional locks employed in each conflict detection strategy, and the bookkeeping time that takes place inside the transactional manager.

Figure 7.6 presents the bookkeeping overhead of each rollback mechanism for writes within transactions that end up either committing or aborting. In the case of a commit, the graph clearly shows the additional cost of operating a hashtable as opposed to a
sequential buffer when implementing write-buffering.

The advantage of write-buffering over undo-logging in cases with a high abort rate is also illustrated. We can see that, even when both rollback mechanisms are implemented with sequential buffers, the overhead incurred by the transaction manager using write buffering is significantly smaller. This is because with undo logging the manager has to make a copy when first accessing the location and then another one to restore the original value at the time of the abort. This is not the case for write-buffering, where the manager discards the buffers if the transaction aborts.

Figure 7.7 presents the bookkeeping overhead of each conflict resolution strategy for read operations. The results show the penalty paid by the abort readers policy, which require an additional copy into a temporary location followed by an abort status check. This occurs whenever a location is read to make sure that the current thread hasn’t been aborted while performing its access.

Figure 7.8 presents the locking overhead associated with every conflict detection strategy implemented. All strategies present similar results when it comes to managing read accesses. However, for write operations we can see that the read-optimistic strategies incur double the overheads of other conflict detection strategies. This can be explained by the fact that these strategies require two atomic operations: one for acquiring the write lock and another for establishing exclusiveness against other readers. The pessimistic and the fully-optimistic strategies perform these actions in one step and as a result require only one atomic operation.
Chapter 8

Related Work

Shavit and Touitou [25] proposed the first software TM (STM). Their design was non-blocking and its main disadvantage was that it supported only static transactions. Later proposals of non-blocking schemes by Herlihy et al. [14] in dynamic STM (DSTM) and by Fraser and Harris [9] in object-based STM (OSTM) addressed this problem. OSTM is able to provide lock freedom guarantees by using a conflict resolution strategy called helping. In order to guarantee progress, transactions help those ahead of them, but that was only possible at the cost of maintaining public shared transaction records. A more efficient implementation of a non-blocking STM, called RSTM, was proposed by Marathe et al. [18]. As opposed to DSTM, its metadata organization uses only one level of indirection, thereby lowering its costs related to bookkeeping. Based on this implementation, eager, lazy and mixed approaches to conflict detection have also been investigated by Spear et. al. [27], with the authors concluding that none of them works best across all workloads.

Ennals argued in his study [7], that most of the properties associated with non-blocking algorithms are either unnecessary or can be replaced with other mechanisms provided by modern operating systems. For instance, the fact that non-blocking implementations of STM libraries continue to scale, even when running a number of threads
Chapter 8. Related Work

larger than the number of processors, could also be achieved by using the schedctl() system call available on Solaris. A similar approach was taken by Adl-Tabatabai et al. and Saha et al. [1, 23] in their compiler and runtime framework for supporting efficient software transactional memory. Both proposal provide consistency by using an extension of the two phase locking algorithm with pessimistic conflict detection. Dice and Shavit suggest in [5], that both Ennals and Saha failed to observe that while pessimistic strategies have an advantage due to lower overheads, they perform poorly for data structures under high contention. In this type of scenarios a pessimistic approach is more likely to livelock, due to the fact that locks are held for a longer duration than in the case of an optimistic strategy.

The use of timestamp-based ordering protocols has also been extensively explored. Different proposals use distinct conflict detection strategies ranging from pessimistic (e.g. TinySTM [21, 22, 8]), to optimistic (e.g. TL2 [4]), with SwissTM [6] adopting a hybrid approach. Unfortunately, the decision to employ one policy over another is largely based on empirical evidence. In contrast, in our work we identify the exact scenarios where a policy outperforms the others, and provide a mechanism for making sure that the fully-optimistic strategy provides the best performance independent of the conflict pattern exhibited by the application.

Finally, in [2] Aydonat and Abdelrahman relax the requirements of linearizability, and implement a system based on conflict-serializability. Additionally, multi-versioning is used in order to allow for more concurrency between otherwise conflicting transactions. However, we argue that this system could still benefit from our approach to conflict detection, as it would minimize the cost of enforcing an ordering between two transactions that requires one of them to abort.

Contention managers have been proposed as another mechanism for mitigating conflicts between transactions in workloads that exhibit high contention. This option has been studied by Scherer and Scott [16] and Guerraoui et al. [11, 10]. However, their main
focus has been on providing progress and fairness guarantees rather than exploring the trade-offs of conflict detection and resolution strategies, per se. Contention managers are oblivious to the type of conflicts that they are trying to resolve and their ability to intercept conflicts is only as comprehensive as the conflict detection mechanism that they employ. This correlation has been underlined by Dragojević et. al. [6] with a series of experiments showing that whether a conflict management scheme achieves its full potential is dependent on the conflict detection policy used, with some policies being better suited for a specific contention management scheme than others.

Koskinen et al. [17] explore the syntactic benefits of using checkpoints instead of nested transactions in conjunction with a semantic-based transaction manager. In contrast, our work outlines the implications of using partial rollbacks at the level of the conflict detection strategy, mainly by removing the risks associated with optimistic policies. Moreover, their approach requires the user to provide application-specific mechanisms for deciding how far back to revert a partially aborted transaction, whereas the design of our library makes this step seamless to the programmer.

Waliullah and Stenstrom [28] proposed, in the context of a hardware transactional memory, a history-based mechanism that would monitor the locations that are most contented and record checkpoints prior to accessing them. However, such a solution does not take into account the dynamic nature of some workloads, where objects are constantly created and deleted from the workset.
Chapter 9

Conclusions

We performed a systematic study of all the conflict scenarios that may emerge between two transactions. We observed that the optimum conflict detection policy varies from one conflict pattern to another, and we correlated application conflict patterns with the type of conflict detection strategy which provides the best performance. We validated this analysis by implementing and evaluating a TM prototype, *libTM*, where the type of conflict detection can be varied based on the type of conflict being detected, resulting in a range of conflict detection policies, from the fully-pessimistic to the fully-optimistic.

We showed the effectiveness of providing support for partial rollbacks in mitigating inherent conflict detection policy pit-falls and providing stable performances close to the best approach in all cases. By allowing transactions to perform partial rollbacks we were successful in alleviating the risks of wasted work associated with optimistic protocols.

With an unified approach to conflict detection, one of the major variables of concurrency control is eliminated, opening the door to more precise design and evaluation of the other major factor influencing performance: contention management schemes.

Finally, we also analyzed the overheads incurred by each layer in the implementation of our library, as well as the restrictions imposed by the conflict detection strategies on the mechanisms supporting recoverability that may be employed.
Bibliography


