On the Input Acceptance of Transactional Memory

Vincent Gramoli
EPFL, University of Neuchâtel

Derin Harmanci
University of Neuchâtel

Pascal Felber
University of Neuchâtel

Abstract

We present the Input Acceptance of Transactional Memory (TM). Despite the large interest for performance of TMs, no existing research work has investigated the impact of solving a conflict that does not need to be solved. Traditional solutions for a TM to be correct is to delay or abort a transaction as soon as it presents a risk to violate consistency. Both alternatives are costly and should be avoided if consistency is actually preserved. To address this problem, we introduce the input acceptance of a TM as its ability to commit transactions, we upper-bound the input acceptance of existing TMs and propose a new TM with higher input acceptance.

1 Introduction

Transactional Memory (TM) has recently been proposed as a parallel programming paradigm to take benefit of upcoming multicore architectures. In contrast with the lock-based paradigm, TM uses speculative execution of transactions for simplicity reasons: semantics is preserved under transaction composition. In TM systems, transactions are scheduled in parallel on distinct threads as sequences of transactional operations. Due to this parallelism, operations of concurrent transactions accessing a common shared object can naturally be interleaved. Some TMs require conflict resolution if at least one of these operations is a write [8], others, however, require conflict resolution only if the first occurring operation is a write [1]. Roughly speaking, a conflict represents a risk that the TM consistency be violated. Common conflict resolution consists either in forcing one of the two conflicting threads to sleep until the other terminates executing its transaction, or in forcing one of the two transactions to abort for later restart.

Clearly, forcing a thread to sleep may imply that the core executing this thread remains idle. Moreover, this might not successfully resolve the conflict. Multicore architectures are inherently parallel and all cycles during which a core

remains idle are wasted. As a result, resuming one transaction after the other, would possibly resolve conflicts but it would not exploit mutlicore resources efficiently. Unlike database transactional systems where transactions are buffered on the server-side and could preferably be executed sequentially [5], multicore architectures greatly benefit from concurrent executions of transactions. Here, we focus on transactional memory systems that fully exploit multicore architecture. In other words, we aim at minimizing the idle time of each core, that is, we focus on non-delaying contention managers.

Nevertheless, if all operations are executed without being postponed, then solving a conflict requires to abort one of the two conflicting transactions. Aborting a transaction implies to roll-back the operations executed in this transaction and to restart it later, hence, aborting may be considered as a waste of efforts as well.

Figure 1: An input pattern for which numerous STMs try unnecessarily to resolve a conflict. In this example, we chose a contention manager that aborts the transaction detecting the conflict.

In the example of Figure 1, two transactions execute concurrently so that their operations are interleaved. The read operation at thread p_2 could return indifferently the new value of x or the overwritten value without violating consistency. Aborting or delaying this transaction is thus unnecessary since committing it would not violate serializability [14], opacity [9], or even linearizability [12]. Interestingly, many Software Transactional Memories (STMs) unnecessarily try to resolve a conflict for the sake of simplicity [3, 4, 8, 10, 11]. In this paper, the goal is to minimize the number of unnecessary aborts while fully exploiting cores.

Contributions. This paper introduces the input acceptance of transactional memories as a measurement of their ability to commit transactions. We identified five designs shared by seven TMs and compared their input acceptance upper-bound. Upper-bound stands here for the limited amount of input the design accepts: the more input it accepts, the higher the upper-bound. The resulting design classification is confirmed experimentally on realistic workloads. Here are our designs:

1. Visible read (VWVR): this design used for instance by SXM [8] let the other threads know of a read operation immediately after the corresponding read request is received (visibility is ensured by setting a flag or locking a variable);

2. Visible write (VWIR): this design used for instance by DSTM [11] and TinySTM [4] makes the effect of a write operation visible to other threads immediately after the corresponding write request is received;

3. Invisible write (IWIR): this design used by WSTM [10] and by TL2 [3] delays the effect of a write operation until reception of the commit request of the same transaction (neither reads nor writes are made visible before commit-time);

4. Commit-time relaxation (CTR): this design used in TSTM [1] allows to order transactions independently from the time a commit request is received;

5. Real-time relaxation (RTR): this design relaxes the constraint that if a transaction t_1 ends before another transaction t_2 starts, then all the operations of t_1 must precede operations of t_2.

We propose a Serializable Software Transactional Memory, namely *SSTM*, that implements the last design. SSTM presents a higher input acceptance than other STMs and does not suffer from congestion since it uses shared object metadata instead of global parameters to detect conflicts.

Related work. The question whether a set of input transactions can be accepted without being rescheduled has already been studied by Yannakakis [17]. Similarly to our work, this paper considers that the scheduler receives the workload and reschedules it into a sequentially-equivalent output. More precisely however, this paper focuses on the expressiveness of concurrency, and does not take into account TM constraints. In contrast here, we especially concentrate on TMs where some operation requests must be treated immediately for efficiency reasons.

The recent *permissiveness* property [7] measures the variety of comitted outputs. Unfortunately, permissiveness does not capture the amount of workloads TMs accept: even if a single workload is accepted the TM can be considered as highly permissive if it produces a large variety of safe histories. That is, a TM can have a very high permissiveness with a very low input acceptance. Similarly to the commit-abort ratio, [15] introduces the *abort-rate* but not to compare STMs.

Some STMs present desirable features that we also target in this paper. All these STMs relax a requirement common to opacity and linearizability to accept a wider set of workloads: the real-time order. As far as we know SSTM is, however, the first of these STMs that is fully decentralized and ensures serializability. CS-STM [16] is decentralized but is not serializable. Existing serializable STMs require either centralized parameters [13] or a global reader table [1] to minimize the number of aborting transactions.

The rest of the paper is organized as follows. Section 2 presents the model and some preliminary definitions. Section 3 introduces TM designs and input

classes, and upper-bounds the input acceptance of TM designs. Section 4 shows the correctness of SSTM, our high input acceptance STM. Section 5 compares the input classes and Section 6 validates this generalization experimentally. Finally, Section 7 concludes the paper.

2 Model and Definitions

A TM *execution* takes as input a workload and produces an associated history that satisfies consistency. This section formalizes the notions of workload and history as TM input and TM output, respectively. In our model, we assume that all input events are part of a transaction and that no transactions are nested. We also assume that when a transaction aborts it must be retried later—the retried transaction is then considered as a distinct one.

2.1 TM Input

First, we introduce TM input as a formalization of the notion of workload. An *input event* is either a start request, an operation call on a shared variable, or a commit request. Here, we only admit read and write operations and all operations are part of a transaction. We denote a start request, a read call on x, a write call on x, and a commit request as part of the same transaction t by s_t, $r(x)_t$ (or r_t^x for short), $w(x)_t$ (or w_t^x for short), and c_t. The values read and written are of no interest in the input definition and they are omitted from the notations of input events. We use π_t to refer indifferently to a read or a write operation: either r_t or w_t.

An *input pattern* $\mathcal{P}$ of a TM is a (totally ordered) sequence of input events. The associated order corresponds intuitively to the real-time order in the sense that one event is ordered before another if and only if its execution precedes the other in time, and for the sake of simplicity we assume that no two distinct events occur at the same time. Observe that this assumption is reasonable since two operations on the same shared variable will be ordered by the TM (e.g., using a compare-and-swap) and non-conflicting concurrent events can be arbitrarily ordered. An input pattern is *well-formed* if each event $\pi(x)_t$ of this pattern is preceded by a unique s_t and followed by a unique c_t. An *input class* $\mathcal{C}$ can be a set of input patterns (potentially infinite). An *input transaction* executed by thread (or processor) p refers to a sub-pattern of the input composed of all events between a start request and the first following commit request c applied to thread p (both start and commit are included).

2.2 TM Output

Second, we define TM output as the classical notion of history. This history is produced by the TM as a result of a given input. An *output event* is a read or write operation that has returned, a commit, or an abort. We refer to the read operation of transaction t that accesses shared variable x and returns value v_0,

as $R(x)_t : v_0$. Similarly, we refer to a write operation of t writing value v_1 on variable x as $W(x, v_1)_t$. In the output definition, written values are necessary to decide upon the output correctness. We refer indifferently to Π_t as either a read operation or a write operation executed by t, and to C and A as a commit and abort, respectively. A *history* H of a transactional memory is a pair $\langle O, \prec \rangle$ where O is a set of output events and $\prec$ is a total order defined over O. A *projection* of a history H on a thread p is a sub-history $H_p = \langle O_p, \prec \rangle$ where O_p is the set of all events of O executed by thread p. We omit the operation subscript t and the history subscript p when the associated thread and transaction are clear from the context.

As mentioned earlier, the ordering $\prec$ corresponds simply to the real-time precedence of the instants at which the events occur. For short, we say that an operation Π_1 "precedes" another operation Π_2 if and only if $\Pi_1 \prec \Pi_2$ and we assume that any two distinct events occur at distinct time instants. Observe that the order given by single-threaded execution is included in the real-time order: the former implies the latter. An *output transaction* executed by thread p is a sub-history of H_p composed of all events between a commit/abort (excluded) or the first event of H (included) and the first following commit/abort event (included). For each input transaction t, there exists exactly one associated output transaction t' whose sequence of operations results from a subsequence of operation requests of t and that commits or aborts. More precisely, an execution is *well-formed* if *(i)* the input pattern is well-formed, *(ii)* there is a one-to-one mapping from the input transactions to the output transactions, *(iii)* each output transaction is either the sequence of events resulting from its mapped input transaction, or the sequence of events resulting from a prefix of its mapped transaction plus an abort event.

By abuse of notation, we refer indifferently to a transaction as an input transaction or its associated output transaction.

2.3 Consistency

Two operations π_1 and π_2 *conflict* if and only if *(i)* they are part of different transactions, *(ii)* they access the same variable x, and *(iii)* at least one of them is a write operation. We denote a conflict by $\pi_1 \longrightarrow \pi_2$ if π_1 precedes π_2 with respect to the sequential specification of variable x, i.e., π_2 reads the value of x written by π_1, π_2 overwrites the value of x read by π_1, or π_2 overwrites the value of x written by π_1. (Otherwise, if π_2 precedes π_1 with respect to the sequential specification of x, then the conflict is denoted by $\pi_2 \longrightarrow \pi_1$.) A transaction t_1 *precedes* a transaction t_2 if and only if π_1 and π_2 are operations of t_1 and t_2, respectively, and there is a conflict $\pi_1 \longrightarrow \pi_2$.

A *complete history* is a history where all events are part of a committed transaction, i.e., a transaction whose last event is C. Hence, no transactions are unfinished or aborted in a complete history. The complete history $C(H)$ of H is the history H where all events that are not part of a committed transaction has been removed.

A transaction t_1 precedes a transaction t_2 if and only if π_1 and π_2 are operations of t_1 and t_2, respectively, and there is a conflict $\pi_1 \longrightarrow \pi_2$. We denote this *precedence relation* by $t_1 \xrightarrow{W} t_2$ if π_1 is a write operation and by $t_1 \xrightarrow{R} t_2$ if π_1 is a read operation, or indifferently by $t_1 \longrightarrow t_2$. We refer to a *path* p as an ordered sequence of precedences between transactions: $p = t_1 \longrightarrow t_2 \longrightarrow ... \longrightarrow t_k$. A serializability graph of a history H is the graph $SG(H)$ whose nodes are the committed transactions of H and where an edge exists between transactions t_1 and t_2 if and only if $t_1 \longrightarrow t_2$. A history H is *conflict-serializable* if and only if its serializability graph of its complete history $C(H)$ is acyclic. By extension, a TM is conflict-serializable if and only if it outputs only conflict-serializable histories.

2.4 Classification

An input is composed of a set of events that are totally ordered. Therefore, we can consider an input pattern as a word whose alphabet contains events and an input class as a language defined over the alphabet of possible events. We use regular expressions to represent the possible input patterns of a class. In our regular expressions, parentheses, '(' and ')', are used to group a set of events. The star notation, '$*$', indicates the Kleene closure and applies to the preceding set of events. The complement operator, '$\neg$', indicates any event except the following set. Finally, the choice notation, '$|$', denotes the occurrence of either the preceding or the following set of events. Operators are ordered by priority as $\neg, *, |$.

2.5 Commit-Abort Ratio

The *commit-abort ratio*, denoted by τ, is the ratio of the number of committing transactions over the total number of complete transactions (committed or aborted). This metric captures the notion of success of a TM by giving the percentage of transactions that the TM committed versus the total number of transactions the TM attempted to commit. That is, the commit-abort ratio is an important measure of "achievable concurrency" for TM performance, especially from a theoretical point-of-view.

Throughput is a metric of performance traditionally used in TM to measure the number of transactions a TM commits per time unit. Throughput is, however, not sufficient to identify the cause of TM efficiency: one TM may be efficient either because it aborts very few transactions or because it retries transactions very rapidly. The commit-abort ratio is complementary to the throughput since it determines whether a TM is simply fast or whether it has a high input acceptance. Evaluating how likely a TM aborts transactions is a crucial issue since aborting can be very costly. First, this cost depends on the efforts wasted in executing the transaction before aborting it: typically, a long transaction will be generally costly to retry. Second, abort side-effects might be dramatic for performance: take, as an example, an aborting transaction that

has previously forced several other transactions to also abort, this transaction may create further conflicts upon retry.

In the remaining of the paper, we say that a TM *accepts* an input pattern if it commits all of its transactions, i.e., $\tau = 1$. More generally, we say that a TM *does not accept* an input class if it accepts no pattern of this class. In other words, the TM does not accept a class if for each of its patterns, the TM aborts at least one transaction, i.e., $\tau < 1$.

3 The Input Acceptance of TM Designs

This section identifies several TM designs and upper-bounds their input acceptance. As said earlier, upper-bound stands here for the limited amount of input the design accepts: the more inputs it accepts, the higher the upper-bound. All the designs considered here are non-blocking (no transactions wait for a conflict to possibly disappear) and there is at most one version for each shared variable.

The TM designs that we consider always provide a consistent view of the memory to the application and guarantee sequentially consistent executions (serializability). They may or may not be linearizable: this is typically not important from an application programmer's perspective (although it has some impact on the implementation of the TM).

For each of these designs, we define one input class capturing a set of patterns that are not accepted (although these patterns are accepted by subsequent designs), hence giving an upper-bound of the input acceptance of each design. For the sake of clarity of the design presentations, we assume in the pseudocode of the algorithms that each function is atomic and we do not specify how shared variables are updated. Typical solutions include compare-and-swap [11] or in-order lock acquisition [10]. We refer to T as the set of transaction identifiers, to X as the set of all variable identifiers, and to V as the set of possible variable values.

3.1 VWVR Design

This section introduces a TM design with visible writes and visible reads, called *VWVR*, and shows its acceptance limitation by defining a class of input patterns that this design never accepts. The pseudocode is given in Algorithm 1 and is similar to SXM [8]. For simplicity of presentation, we assume that variables are versioned.

If a read request is input, the TM records the transaction in $x.readers$ (Line 14), thus, the set of variables read is visible to all threads. Similarly, the write operations are made visible in that when a write request is input the updating transaction registers itself in $x.writer$ (Line 24).

It turns out that common input patterns are not accepted by this design. For a classical example of write-after-read pattern by two transactions, consider the example proposed in Figure 2. If a transaction t_2 writes a variable that has already been read by another transaction t_1 that is still active, then a conflict

Algorithm 1 VWVR Design

```
1:  State of transaction t:
2:      read-set ⊂ X, initially ∅
3:      write-set ⊂ X × V, initially ∅

4:  State of shared variable x:
5:      val ∈ V, initially default value
6:      writer ∈ T, initially ⊥
7:      readers ⊂ T, initially ∅

8:  read(x)_t:
9:      if ⟨x, v'⟩ ∈ write-set then v ← v'
10:     else
11:         if x.writer ≠ ⊥ then abort()
12:         v ← last committed value of x
13:         read-set ← read-set ∪ {x}
14:         x.readers ← x.readers ∪ {t}
15:     return v

16: write(x, v)_t:
17:     if x.readers \ {t} ≠ ∅ then abort()
18:     if x.writer = t then
19:         write-set ← (write-set \ {⟨x, ∗⟩}) ∪
20:             {⟨x, v⟩}
21:     else
22:         if x.writer ≠ ⊥ then abort()
23:         write-set ← write-set ∪ {⟨x, v⟩}
24:         x.writer ← t

25: commit()_t:
26:     for each ⟨x, v⟩ ∈ write-set do
27:         x.val ← v
28:         x.writer ← ⊥
29:     for each x ∈ read-set do
30:         x.readers ← x.readers \ {t}

31: abort()_t:
32:     for each ⟨x, v⟩ ∈ write-set do
33:         x.writer ← ⊥
34:     for each x ∈ read-set do
35:         x.readers ← x.readers \ {t}
```

is detected by t_2 while writing. This leads to resolving the conflict. As stated in the following theorem, an input class including this pattern is not accepted by this design.

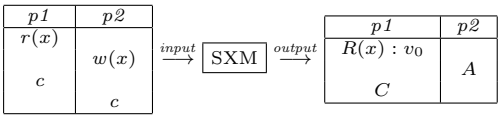

Figure 2: An input pattern for which SXM produces a commit-abort ratio of $\tau = 0.5$ (transaction of *p2* aborts upon writing).

Theorem 3.1 *There is no TM implementing VWVR design that accepts any input pattern of the following class:*

$$\mathcal{C}1 = \pi^*(\pi_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* \pi_i^x)\pi^*, \text{ for any } i \neq j.$$

Proof. The proof of this impossibility relies on the existence of two sub-patterns, of which at least one is common to any pattern of class $\mathcal{C}1$ and that is not accepted by any VWVR STM. Consider the input pattern $\mathcal{P}1 = \pi(x)_1 w(x)_2$ and $\mathcal{P}1' = w(x)_1 \pi(x)_2$.

First, since a write operation on variable x verifies that neither a write operation nor a read operation is accessing x and aborts a transaction if this

verification fails, $\mathcal{C}1$ does not accept $\mathcal{P}1$. Second, since both read and write operations on variable x verify that x is not currently written and abort a transaction if the verification fails, $\mathcal{C}1$ does not accept $\mathcal{P}1'$. That is, neither $\mathcal{P}1$ nor $\mathcal{P}1'$ are accepted by $\mathcal{C}1$.

Finally, observe that adding any event to $\mathcal{P}1$ or $\mathcal{P}1'$ produces a pattern of $\mathcal{C}1$ that is not accepted by VWVR STMs for the same reason as above. As a result, class $\mathcal{C}1$ is not accepted by VWVR STMs. □

3.2 VWIR Design

Next, we introduce a TM design with visible writes and invisible reads, called *VWIR*, that is similar to DSTM [11] and TinySTM [4] with a contention manager that aborts the transaction detecting a conflict. The limitations of this design are shown by giving a class of inputs that it never accepts. The pseudocode is given in Algorithm 2 and presents functions similar to the previous algorithm except that we specify additionally the function validate. If a read request is input, the TM records locally the opened read variable, thus, the set of variables read is visible only to the current thread. Conversely, the write operations are made visible in that when a write request is input the updating transaction registers itself in $x.writer$ (Line 22).

Algorithm 2 VWIR Design

```
1:  State of transaction t:
2:    read-set ⊂ X, initially ∅
3:    write-set ⊂ X × V, initially ∅

4:  State of shared variable x:
5:    val ∈ V, initially default value
6:    writer ∈ T, initially ⊥

7:  read(x)_t:
8:    if ⟨x, v'⟩ ∈ write-set then v ← v'
9:    else
10:     if x.writer ≠ ⊥ then abort()
11:     validate()
12:     v ← last committed value of x
13:     read-set ← read-set ∪ {x}
14:   return v

15: write(x, v)_t:
16:   if x.writer = t then
17:     write-set ← (write-set \ {⟨x, ∗⟩}) ∪
18:       {⟨x, v⟩}
19:   else
20:     if x.writer ≠ ⊥ then abort()
21:     write-set ← write-set ∪ {⟨x, v⟩}
22:     x.writer ← t

23: commit()_t:
24:   validate()
25:   for all ⟨x, v⟩ ∈ write-set do
26:     x.val ← v
27:     x.writer ← ⊥

28: abort()_t:
29:   for all ⟨x, v⟩ ∈ write-set do
30:     x.writer ← ⊥

31: validate()_t:
32:   for all x ∈ read-set do
33:     x' ← last committed version of x
34:     if x ≠ x' then abort()
```

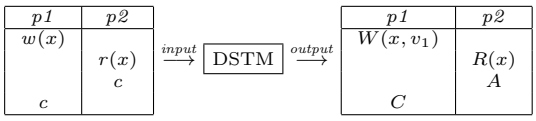

Figure 3: A simple input pattern for which DSTM produces a commit-abort ratio of $\tau = 0.5$ (transaction of *p2* aborts).

Common input patterns are not accepted by this design. Consider the input pattern depicted in Figure 3 that may arise for instance when concurrent operations (searches, insertions) are executed on a linked list.

This is a classical example of read-after-write pattern by two transactions, with the written value being visible and uncommitted. If a transaction t_2 reads a variable previously modified by another transaction t_1 that is still active, then a conflict is detected by t_2 while reading. In any case, this leads to resolving the conflict: while in this design the transaction t_2 aborts due to this conflict, any alternative contention manager aborts one of the current transactions.[1] As stated in the following theorem, an input class including this pattern is not accepted by this design.

Theorem 3.2 *There is no TM implementing VWIR design that accepts any input pattern of the following class:*

$$\mathcal{C}2 = \pi^*(r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \mid w_j^x \neg c_j^* r_i^x)\pi^*, \text{ for any } i \neq j.$$

Proof. The proof is similar to the proof of Theorem 3.1 but with the following patterns: $\mathcal{P}2 = r(x)_1 w(x)_2 c_2$ and $\mathcal{P}2' = w(x)_1 r(x)_2$.

Since in $\mathcal{P}2$, t_2 writes and commits the value of x after the time at which t_1 reads x and before the time at which t_1 commits, t_1 fails in validating right before commit-time and aborts. As a result, $\mathcal{P}2$ is not accepted by $\mathcal{C}2$. Since in $\mathcal{P}2'$, t_2 reads the value of x after the time at which t_1 writes x and before the time at which t_1 commits, the read operation fails because t_2 knows that t_1 is still the writer of the object. As a result, $\mathcal{P}2'$ is not accepted by $\mathcal{C}2$.

Next, observe that adding events to $\mathcal{P}2$ or $\mathcal{P}2'$ results in a pattern of $\mathcal{C}2$ that is not accepted by VWIR STMs for the same reason as above. □

As mentioned earlier, this input class captures realistic workloads composed of common read and update transactions.

3.3 IWIR Design

Here, we propose a third design that accepts patterns of the preceding classes, i.e., for which the previous impossibility results do not hold. Nevertheless, we

[1] Observe that the algorithm could be extended to detect read-only transactions, allowing the transaction of thread $p2$ to commit in this specific scenario. In the general case, however, one of the transactions will abort.

do not claim that all patterns of $\mathcal{C}1$ or $\mathcal{C}2$ are accepted by this design. This design, inspired by WSTM [10] and TL2 [3], uses invisible writes and invisible reads with a lazy acquire technique that postpones effects until commit-time, thus it is called *IWIR*. While a main constraint of TMs is that a read must return without being postponed, TMs allow us to postpone a write operation, thus delaying its visibility. The idea differs from the previous designs due to the invisibility of writes: while modifications are recorded at write-time in the *write-set*, these modifications are made visible not earlier than commit-time. The corresponding functions and states are presented in Algorithm 3.

Algorithm 3 IWIR Design

```
1:  State of transaction t:
2:      read-set ⊂ X, initially ∅
3:      write-set ⊂ X × V, initially ∅
4:  State of shared variable x:
5:      val ∈ V, initially default value
6:  read(x)_t:
7:      if ⟨x, v'⟩ ∈ write-set then v ← v'
8:      else
9:          validate()
10:         v ← last committed value of x
11:         read-set ← read-set ∪ {x}
12:     return v
13: write(x, v)_t:
14:     write-set ← (write-set \ {⟨x, ∗⟩}) ∪
15:         {⟨x, v⟩}
16: commit()_t:
17:     validate()
18:     for all ⟨x, ∗⟩ ∈ write-set do
19:         x' ← last committed version of x
20:         if x ≠ x' then abort()
21:     for all ⟨x, v⟩ ∈ write-set do
22:         x.val ← v
23: abort()_t: —
24: validate()_t:
25:     for all x ∈ read-set do
26:         x' ← last committed version of x
27:         if x ≠ x' then abort()
```

Even the IWIR design does not accept some very common input patterns, as mentioned in the introduction and as depicted in Figure 1. This is a classical example of transaction writing a value that is later read. Such a pattern arises, for example, when performing concurrent operations on a linked list. The following theorem gives a set of input patterns that are not accepted by TMs of the IWIR design.

Theorem 3.3 *There is no TM implementing IWIR design that accepts any input pattern of the following class:*

$$\mathcal{C}3 = \pi^*(r_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* r_i^x) \neg c_i^* c_j \pi^*, \text{ for any } i \neq j.$$

Proof. In this proof we consider the following two patterns $\mathcal{P}3 = r(x)_i w(x)_j c_j$ and $\mathcal{P}3' = w(x)_j r(x)_i c_j$ of $\mathcal{C}3$. We show that each of these patterns is not accepted.

First, consider the input pattern $\mathcal{P}3$, and assume by contradiction that its two transactions commit. Upon invocation of $r(x)_i$, transaction i records the variable in its read-set for later validation. At the time t_j commits, the variable

x is updated with the new value written by t_j. Since t_i has not committed yet when the write becomes visible, upon committing, t_i fails in validating its read-set leading to an abort.

Second, consider the input pattern $\mathcal{P}3'$, and assume by contradiction that the two transactions commit. Since writes are invisible and $r(x)_i$ occurs before c_j, the value written by t_j is not read by t_i. That is, $\mathcal{P}3'$ and $\mathcal{P}3$ becomes indistinguishable from t_i standpoint. As above, upon committing, t_i fails in validating leading to an abort.

Clearly, adding any sequence of operations between the three events of $\mathcal{P}3$ and $\mathcal{P}3'$ would lead also to non-accepted patterns. Since all possible patterns of $\mathcal{C}3$ contain one of these two sub-patterns, input class $\mathcal{C}3$ is not accepted by IWIR STMs. □

Note that this impossibility result also holds for the VWVR and VWIR designs, since $\mathcal{C}3$ is a subset of $\mathcal{C}1$ and $\mathcal{C}2$ as we indicate in Section 5.

3.4 CTR Design

The following design has, at its core, a technique that makes as if the commit occurred earlier than the time the commit request was received. In this sense, this design relaxes the commit time and we call it *Commit-Time Relaxation (CTR)*. To this end, the TM uses scalar clocks that determine the serialization order of transactions. The pseudocode appears in Algorithm 4 and is inspired by the recently proposed TSTM [1] in its single-version mode. The first particularity is that a $\mathsf{read}(x)$ request forces the clock of the transaction to be at least as large as the clock of the last transaction that committed x (which also corresponds to the version of x). The second particularity is that committing a transaction t_1 that writes x forces active readers of x to have a clock lower than t_1's. Due to the second particularity, even though a transaction t_2 is not completed yet, an already committed transaction t_1 may force t_2 to be serialized before.

TSTM is claimed to achieve conflict-serializability, however, it does not accept all possible conflict-serializations. Figure 4 (center and left-hand side) presents an input pattern that TSTM does not accept since transactions choose their clock depending on the last committed version of the object they access: in this example, transactions of $p2$ and $p3$ choose the same clock and force $p1$ transaction to abort. This pattern typically happens when a long transaction t runs concurrently with short transactions that update the variables read by t. The following theorem generalizes this result by showing that STMs implementing CTR design does not accept a new input class.

Theorem 3.4 *There is no TM implementing CTR design that accepts any input pattern of the following class:*

$$\mathcal{C}4 = (\neg w^x)^* r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \neg c_i^* s_k \neg (c_i \mid c_k \mid r_k^x)^* w_k^y \neg (c_i \mid c_k \mid r_k^x)^* c_k \neg c_i^* r_i^y \pi^*,$$
for any disctinct i, j, and k.

Algorithm 4 CTR Design

```
1:  State of transaction t:
2:    status ∈ {active, inactive}, initially active
3:    read-set ⊂ X, initially ∅
4:    write-set ⊂ X × V, initially ∅
5:    clock-int, a record with fields:
6:      lb ∈ ℕ, initially 0 // clock range lower bound
7:      ub ∈ ℕ, initially ∞ // clock range upper bound
8:    clock ∈ ℕ ∪ {⊥}, initially ⊥
9:    n ∈ ℕ, the number of threads

10: State of shared variable x:
11:   val ∈ V
12:   clock ∈ ℕ, initially 0
13:   active-readers ⊂ T, initially ∅

14: read(x)_t:
15:   x.active-readers ← x.active-readers ∪ {t}
16:   clock-int.lb ← max(x.clock, clock-int.lb)
17:   if clock-int.ub < clock-int.lb then abort()
18:   read-set ← read-set ∪ {x}
19:   return x

20: write(x, v)_t:
21:   write-set ← (write-set \ {⟨x, ∗⟩}) ∪ {⟨x, v⟩}

22: commit()_t:
23:   for all ⟨x, ∗⟩ ∈ write-set do
24:     clock-int.lb ← max(x.clock, clock-int.lb)
25:     if clock-int.ub ≠ ∞ then
26:       clock ← clock-int.ub
27:       if clock < clock-int.lb then abort()
28:     else
29:       clock ← clock-int.lb + n
30:       if clock > clock-int.ub then abort()
31:     for all r ∈ x.active-readers do
32:       if r.status ≠ active then
33:         x.active-readers ← x.active-readers \ {r}
34:       else
35:         r.clock-int.ub ← clock − 1
36:   for all ⟨x, v⟩ ∈ write-set do
37:     x.clock ← clock
38:     x.val ← v
39:   status ← inactive

40: abort()_t:
41:   status ← inactive
```

Proof. The proof relies on the existence of a sub-pattern $\mathcal{P}4$ common to any pattern of $\mathcal{C}4$ that is not accepted by the CTR design. Let $\mathcal{P}4$ be $r(x)_i w(x)_j c_j s_k w(y)_k r(y)_i$. First, observe that when t_j commits, it chooses clock n, where n is the number of threads and it upper-bounds the clock of t_i to $n-1$. Second, when t_k commits it sets its clock to n so that t_i sets its lower-bound

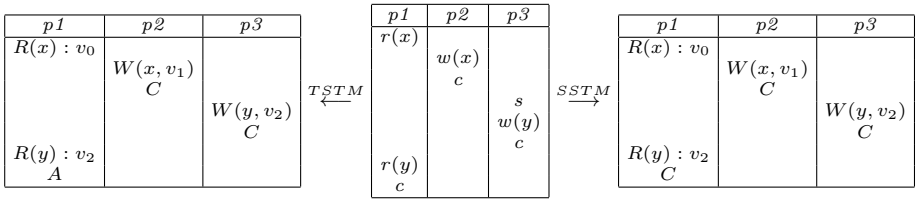

Figure 4: An input pattern (in the center) that TSTM does not accept as described on the left-hand side. The commit-abort ratio obtained for TSTM is $\tau = \frac{2}{3}$ (transactions of *p2* and *p3* commit but transaction of *p1* aborts). In contrast, the Serializable Software Transactional Memory (SSTM) presented in Subsection 3.5 accepts it (the output of SSTM, on the right-hand side, shows a commit-abort ratio of 1).

to n too, when reading y. Consequently, t_i has a larger lower-bound n than its upper-bound $n-1$, that is, t_i aborts upon reading y.

Next, we show that for any other pattern of $\mathcal{C}4$, t_i aborts for the same reason. By the definition of $\mathcal{C}4$, variable x cannot be written before $\mathcal{P}4$ in any pattern of $\mathcal{C}4$. As a result, the upper-bound of t_i cannot be larger than $n-1$. Since t_k does not read, while committing, t_k cannot choose a lower clock than n. Hence, when t_i commits, it sets its lower-bound to n or to a larger value than n, and t_i aborts similarly as above. □

Observe that we use the notation s_k in this class definition to prevent transactions t_j and t_k from being concurrent.

3.5 RTR Design

This design, called *Real-Time Relaxation (RTR)*, presents a technique that relaxes the real-time order requirement. The real-time order requires that given two transactions t_1 and t_2, if t_1 ends before t_2 starts, then t_1 must be ordered before t_2. The design presented here outputs only serializable histories but does not preserve real-time order. More precisely, it outputs non real-time ordered histories as we can see in Figure 4 (center and right-hand side). These outputs result from inputs that cannot be accepted by any TM ensuring real-time order (including all TMs that are opaque [9] or linearizable [12]). We illustrate this design by the following STM.

SSTM, standing for *Serializable STM*, is an STM with a high commit-abort ratio: SSTM accepts all patterns presented so far (including the ones of Figures 1, 2, 3, and 4). Moreover, SSTM is conflict-serializable but neither opaque nor linearizable as shown below, and it avoids cascading abort, since whenever a transaction t_1 reads a value from another transaction t_2, t_2 has already committed [2]. Finally, SSTM is also fully decentralized, i.e., it does not use global parameters as opposed to other serializable STMs [1, 13] that may experience congestion when scaling to large numbers of cores. Figure 5 presents the pseu-

docode of SSTM. As mentioned earlier and like previous designs, functions are assumed to execute atomically for the sake of simplicity in the presentation.

Algorithm 5 SSTM – Serializable Software Transactional Memory

```
1:  State of transaction t:
2:    status ∈ {active, inactive}, initially active
3:    write-set ⊂ X × V, initially ∅
4:    read-set ⊂ X, initially ∅
5:    past-tx ⊂ T, initially ∅ // the previous tx in the conflict graph
6:    future-tx ⊂ T, initially ∅ // the next tx in the conflict graph

7:  State of shared variable x:
8:    write-fc ⊂ T, initially ∅ // the write future conflicts
9:    active-readers ⊂ T, initially ∅ // the active reader tx
10:   val ∈ V, initially the default value

11: write(x, v)_t:
12:   write-set ← (write-set \ {⟨x, ∗⟩}) ∪ {⟨x, v⟩}

13: read(x)_t:
14:   read-set ← read-set ∪ {x}
15:   if ⟨x, v'⟩ ∈ write-set then
16:     v ← v'
17:   else
18:     x.active-readers ← x.active-readers ∪ {t}
19:     for all t' in x.write-fc do
20:       for all t'' ∈ t'.past-tx do
21:         if t = t'' then abort()
22:         past-tx ← past-tx ∪ {t''}
23:       past-tx ← past-tx ∪ {t'}
24:     for all t' in past-tx do
25:       for all t'' ∈ future-tx do
26:         if t' = t'' then abort()
27:         t'.future-tx ← t'.future-tx ∪ {t''}
28:       t'.future-tx ← t'.future-tx ∪ {t}
29:     v ← x.val
30:   return v

31: abort()_t:
32:   status ← inactive
33:   a-clean()
```

During the execution of SSTM, a transaction records the accessed variables locally and registers itself as a potentially future conflicting transaction in the accessed variables. These records help SSTM keeping track of all potential conflicts. More precisely, a transaction t accessing variable x keeps track of all transactions that may both precede it and follow it. Only transactions that read and that are concurrent with t (namely, the active readers of t) can both precede and follow t. This is due to invisible writes that can only be observed by other transactions after commit. When detected, the preceding transactions are recorded in $t.past\text{-}tx$. Transaction t detects those transactions either because they are in $x.active\text{-}readers$ (Line 41) or precede one of these (Line 40),

```
34: commit()_t:
35:   for all ⟨x, v⟩ ∈ write-set do
36:     x.write-fc ← x.write-fc ∪ {t}
37:     for all t′ ∈ x.active-readers ∪ x.write-fc do
38:       for all t″ ∈ t′.past-tx do
39:         if t = t″ then abort()
40:         past-tx ← past-tx ∪ {t″}
41:       if t ≠ t′ then past-tx ← past-tx ∪ {t′}
42:     for all t′ in past-tx do
43:       for all t″ ∈ future-tx do
44:         if t′ = t″ then abort()
45:         t′.future-tx ← t′.future-tx ∪ {t″}
46:       t′.future-tx ← t′.future-tx ∪ {t}
47:   for all ⟨x, v⟩ ∈ write-set do
48:     x.val ← v
49:   status ← inactive
50:   c-clean()

51: a-clean()_t:
52:   for all x such that ⟨x, ∗⟩ ∈ write-set or x ∈ read-set do
53:     x.write-fc ← x.write-fc \ {t}
54:     x.active-readers ← x.active-readers \ {t}
55:   for all t′ ∈ past-tx do
56:     t′.future-tx ← t′.future-tx \ {t}
57:   for all t′ ∈ future-tx do
58:     t′.past-tx ← t′.past-tx \ {t}
59:   free(t)

60: c-clean()_t:
61:   for all x such that ⟨x, ∗⟩ ∈ read-set do
62:     x.active-readers ← x.active-readers \ {t}
63:   for all t′ ∈ T do
64:     if t′.status = inactive and t′.past-tx = ∅ then
65:       past-tx ← past-tx \ {t′}
66:       for all t″ ∈ t′.future-tx do
67:         t″.past-tx ← t″.past-tx \ {t′}
68:       for all x such that ⟨x, ∗⟩ ∈ t′.write-set do
69:         x.write-fc ← x.write-fc \ {t′}
70:       free(t′)
```

or because they are in *x.write-fc* (Lines 23 and 41) or precede one of these (Lines 22 and 40). Transaction t also keeps track of its succeeding transactions in *t.future-tx* so that it can inform them as soon as it discovers a new preceding transaction. Hence, each transaction t' keeps up-to-date records of t'.*past-tx* and t'.*future-tx*. Transaction t may abort for two reasons. First, if it appears to precede itself in the conflict graph (Lines 21 and 39). Second, if there exists a transaction that t precedes but that also precedes t (Lines 26 and 44). Finally, the a-clean function aims at garbage collecting all metadata associated with the current transaction if it aborts whereas the c-clean functions garbage collect only the metadata corresponding to the past committed transactions that have

nothing in their past, as it is sure these transactions will not create a cycle in the conflict graph later.

Tracking all conflicts is known to be a difficult task [9] while it is easy to check linearizability in a composed manner [12], and SSTM may suffer from the induced memory overhead. TSTM presented, however, encouragingly low overhead when tracking a subpart of the conflicts [1] SSTM track. Even though SSTM is not expected to be the fastest STM on today's architectures, we believe that hardware support may help tracking these predominant conflicts in a near future, and its design would benefit from this, as it presents already a higher input acceptance than other designs. As an example, Figure 4 (center and right-hand side) presents an input pattern that SSTM accepts while other STMs that ensure real-time order do not accept. This is illustrated by the non-acceptance of the same pattern by TSTM, in Figure 4 (center and left-hand side).

4 Correctness Proof of SSTM

In this section, we show that SSTM, presented in Subsection 3.5, is conflict-serializable, but neither opaque nor linearizable.

Lemma 4.1 *If there exists a conflict $p = t_0 \longrightarrow t_1$, t_0 and t_1 are both committed and $t_0.$past-tx $\neq \emptyset$ then $t_1 \in t_0.$future-tx.*

Proof. Observe by definition that $t_0 \longrightarrow t_1$ holds only if there is a conflict between t_0 and t_1, and note that $t_0.$*past-tx* $\neq \emptyset$ prevents t_0 from being cleaned. There are two cases to consider whether the conflicting operations of t_0 is a write. Without loss of generality let x be the common location on which both transactions conflict.

First if t_0 writes x and commits, then t_0 adds itself to $x.$*write-fc* at Line 36. Hence, if t_1 reads x afterwards, then it inserts t_0 in its $t_1.$*past-tx* set at Line 23 and symmetrically inserts itself in $t_0.$*future-tx* at Line 28. Otherwise, if t_1 writes x afterwards, it inserts t_0 in its $t_1.$*past-tx* set at Line 41 and symmetrically adds itself in $t_0.$*future-tx* at Line 46.

Second if t_0 does not write but reads x before t_1 writes x, then t_0 adds itself to $x.$*active-reader* at Line 18 so that t_1 adds it to $t_1.$*past-tx* at Line 41. Again symmetrically, t_1 inserts itself into $t_0.$*future-tx* at Line 46. The result follows. □

The next lemma shows that the relation, defined by set $t.$*future-tx*, between t and the transactions it contains is transitive. Transitivity is necessary to show that a cycle in the conflict graph exists only if a transaction t is in its own $t.$*future-tx*.

Lemma 4.2 *Let t_0, t_1, t_2 be three committed transactions. If $t_2 \in t_1.$future-tx and $t_1 \in t_0.$future-tx then $t_2 \in t_0.$future-tx.*

Proof. Let τ and τ' be the times at which the second operation of the conflict between t_0 and t_1 and the second operations of the conflict between t_1 and

t_2 start, respectively. By the assumption of function atomicity, we know that $\tau \neq \tau'$, hence we focus on the two following cases.

In case $\tau' < \tau$, $t_2 \in t_1.\textit{future-tx}$ and $t_1 \in t_2.\textit{past-tx}$ at time τ. Hence, when the conflict between t_0 and t_1 happens by a read (resp. a write) of t_1, t_1 adds not only t_0 in its *past-tx* at Line 23 (resp. at Line 41) and itself to $t_0.\textit{future-tx}$at Line 28 (resp. at Line 46) but also t_2 at Line 27 (resp. at Line 45), which belongs to its $t_1.\textit{future-tx}$, to $t_0.\textit{future-tx}$.

In case $\tau < \tau'$, $t_0 \in t_1.\textit{past-tx}$ at time τ'. Assume t_2 conflicting operation is a read (resp. a write). Transactions t_0, which belongs to $t_1.\textit{past-tx}$, and t_1 are inserted in $t_2.\textit{past-tx}$ at Line 23 (resp. at Line 41), at time τ'. As a result, t_2 inserts itself to the *future-tx* of both t_0 and t_1 at Line 28 (resp. at Line 46). □

Lemma 4.3 *$t \notin t.\textit{future-tx}$.*

Proof. Assume that $t \in t.\textit{future-tx}$ holds, we proceed by contradiction. Transaction t can only be inserted in $t.\textit{future-tx}$ at Line 28 or at Line 46 because neither reaching Line 27 nor Line 45 with $t = t'$ is possible as transaction t would abort prior to that (Lines 26 and 44). As a result, t was already in $t.\textit{past-tx}$ when Line 28 or 46 has been reached.

Now we show that t cannot be inserted in $t.\textit{past-tx}$ leading to the contradiction. If t already belongs $t \in x.\textit{write-fc}$, then this means that t is executing its commit and all its read operations are past, hence, there is no chance that t can be added to $t.\textit{past-tx}$ during its read operation. Finally, during the execution of a write operation *past-tx* remains unchanged, and during the execution of the commit t cannot be inserted into $t.\textit{past-tx}$ because $t = t'$ (Line 41). □

The following corollary shows that for any history H of SSTM there is no cycle in the serialization graph $SG(H)$ of committing transactions.

Corollary 4.4 *In all histories H of SSTM, there is no path $p = t_1 \longrightarrow ... \longrightarrow t_k \longrightarrow t_1$ such that all t_i commit ($0 < i \leq k$).*

Proof. By absurd, assume that this is possible. We show that this leads to a contradiction. First, by Lemma 4.1 we know that $p = t_1 \longrightarrow ... \longrightarrow t_k \longrightarrow t_1$ implies that $t_{i+1} \in t_i.\textit{future-tx}$ for all i such that $0 < i \leq k-1$ and $t_1 \in t_k.\textit{future-tx}$. Second, by the transitivity property of Lemma 4.2 we obtain that $t_i \in t_i.\textit{future-tx}$ $(0 < i \leq k)$ which contradicts Lemma 4.3. □

Theorem 4.5 *SSTM is conflict-serializable.*

Proof. The proof follows from the conjunction of Corollary 4.4 and Theorem 2.1 of [2]. □

SSTM is conflict-serializable, however, we have not shown yet that SSTM is neither opaque [9] nor linearizable [12].

A simple counter-example is presented in Figure 4 (center and right-hand side). Clearly, the input (center) is accepted by SSTM resulting in the output

on the right-hand side. This output is neither opaque nor linearizable. More precisely, let t_1, t_2, and t_3 be the transactions of p_1, p_2, and p_3, respectively. It is clear that $t_3 \xrightarrow{W} t_1$ and $t_1 \xrightarrow{R} t_2$ implying by transitivity that $t_3 \longrightarrow t_2$, however, because of the real-time precedence requirement common to both opacity and linearizability, $t_3 \not\longrightarrow t_2$ is necessary for the output to be opaque or linearizable. In contrast, this output is equivalent to the sequential execution $t_3 \longrightarrow t_1 \longrightarrow t_2$, thus it is conflict-serializable.

Interestingly, an opaque STM would have to handle multiple versions by memory location to accept the same input, which requires additional work compared to SSTM. As it requires $t_3 \not\longrightarrow t_2$, t_1 could not return v_2 when reading y but would have to return an older version. This raises the question of the importance of transactions real-time precedence as it may hamper operations real-time precedence.

5 Class Comparison

Section 3 gives some impossibility results on the input acceptance by identifying input classes. Here, we use this classification to compare input acceptance of TM designs: if all patterns of a class $\mathcal{C}$ belong also to another class $\mathcal{C}'$, then designs that do not accept $\mathcal{C}'$ neither accept $\mathcal{C}$.

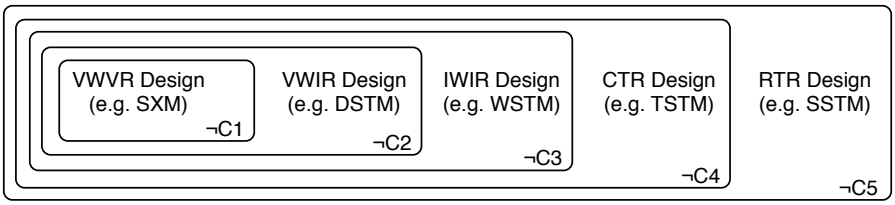

Figure 5: Hierarchization of classes. The VWVR design accepts no input patterns of the presented classes, the VWIR design accepts inputs that are not in classes ranging from $\mathcal{C}2$ to $\mathcal{C}4$, the IWIR design accepts inputs that are neither in $\mathcal{C}3$ nor in $\mathcal{C}4$, the CTR design accepts input patterns only outside $\mathcal{C}4$. Finally, we have not yet identified serializable patterns not accepted by the RTR design.

Looking at the class definitions, we identify interesting dependencies. Let $\mathcal{C}0 = \pi*$ be a special class that represents all possible patterns, and let $\mathcal{C}5 = \emptyset$ be the empty class. Observe that any pattern of class $\mathcal{C}4$ is also a pattern of classes $\mathcal{C}0$, $\mathcal{C}1$, $\mathcal{C}2$, and $\mathcal{C}3$, and any pattern of class $\mathcal{C}3$ is also a pattern of classes $\mathcal{C}0$, $\mathcal{C}1$, and $\mathcal{C}2$. For instance, as stated in Theorem 3.2, STMs implementing the VWIR design (like DSTM) do not accept $\mathcal{C}2$ but $\mathcal{C}5 \subseteq \mathcal{C}4 \subseteq \mathcal{C}3 \subseteq \mathcal{C}2$, hence DSTM accepts none of classes $\mathcal{C}2$ to $\mathcal{C}5$. To represent that a TM accepts only patterns that are outside a class, we draw the sets $\neg\mathcal{C}1$, $\neg\mathcal{C}2$, $\neg\mathcal{C}3$, $\neg\mathcal{C}4$, and

$\neg\mathcal{C}5$ that represent $\mathcal{C}0 \setminus \mathcal{C}1$, $\mathcal{C}0 \setminus \mathcal{C}2$, $\mathcal{C}0 \setminus \mathcal{C}3$, $\mathcal{C}0 \setminus \mathcal{C}4$, and $\mathcal{C}0 \setminus \mathcal{C}5$, respectively. We omit to represent $\neg\mathcal{C}0$ since according to our definition it would be $\emptyset$.

Given this hierarchy, we are able to draw the input acceptance of VWVR, VWIR, IWIR, CTR, and RTR designs restricted to patterns that are in $\neg\mathcal{C}1$, $\neg\mathcal{C}2$, $\neg\mathcal{C}3$, $\neg\mathcal{C}4$, and $\neg\mathcal{C}5$, respectively. Observe that we do not propose patterns that are not accepted by SSTM since our first goal is to differentiate designs among each other, however, we could think of a non-serializable pattern that would not be accepted by SSTM. The hierarchy shown in Figure 5 compares the input acceptance of the TM designs.

6 Experimental Validation

To validate experimentally the tightness of our bounds on the input acceptance of our TM designs, we have implemented all these designs: VWVR, VWIR, IWIR, CTR and RTR. The design comparison presented in Section 5 relies on upper-bounds of input acceptance. Since we ignore how tight these upper-bounds are, some design lower-bounds may not reflect the obtained comparison. To make sure that we did not omit important classes of input patterns, we validate experimentally our theoretical comparison of TM designs.

We have run an integer set linked-list benchmark widely used to evaluate TMs [1, 4, 11, 15] on an 8-core Intel Xeon machine. Initially, the benchmark inserts 256 elements in the linked-list. Then, each thread starts and executes a series of search and update transactions. An update transaction is alternatively an insert or a delete so that the list size remains roughly constant. All search, insert, and delete of value v parses the list in ascendant order while met values are lower than v. Next, the search returns whether the next value is v, the delete removes the next value if it is v, and the insert adds value v if v is not already the next value.

The first series of experiments, presented in Figure 6 (top), compare the input acceptance under high contention on 8 threads. At first glance, the commit-abort ratio decreases as the update probability increases. As one can expect, a larger update probability increases the probability that two transactions conflict, thus the number of aborts in all designs. Second, we can see that the higher a design in the hierarchy of Figure 5, the higher its commit-abort ratio (thus the higher its input acceptance). This clearly confirms our theoretical results. The commit-abort ratio of design VWVR is close to zero because VWVR aborts preferably small transactions each time a write-after-read pattern occurs. More surprisingly, the update probability affects much less the acceptance of the RTR design than any other design. That is, additional contention produces essentially conflicts unnecessary to resolve.

We have performed a second series of experiments to analyze the scalability of each design. These experiments are similar to the first ones with a fixed update probability of 20% and a variable number of threads. The results are depicted in Figure 6 (bottom). This figure clearly illustrates that the acceptance of RTR design scales well with the number of threads, while the other designs

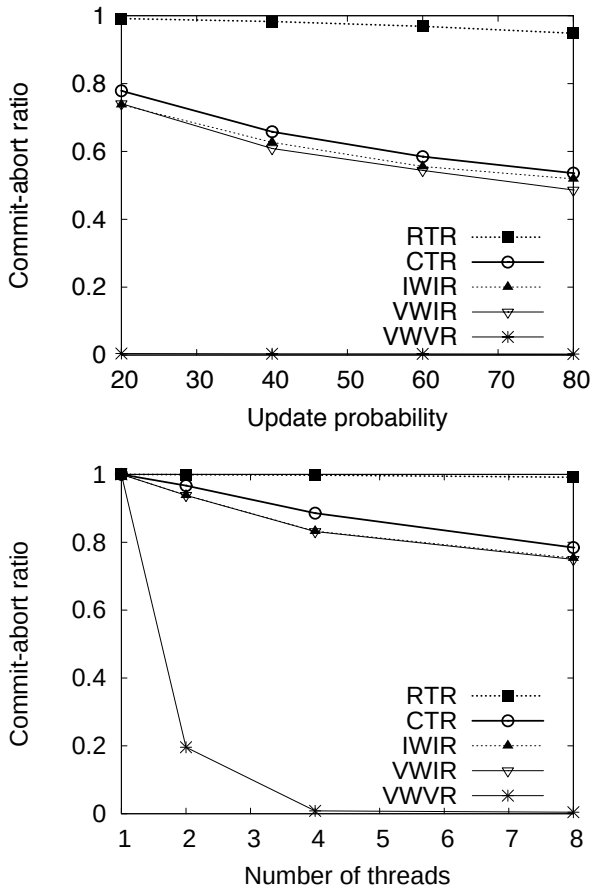

Figure 6: Comparison of average commit-abort ratio of the various designs on a 256 element linked list: (top) with 8 threads as a function of the update probability; (bottom) with a 20% update probability as a function of the number of threads.

have a decreasing acceptance as the number of threads increases. This result indicates how well RTR design copes with conflicts that span transactions of multiple threads.

7 Conclusion

We upper-bounded the input acceptance of well-known TM designs and we proposed a new TM design with a higher acceptance. Preliminary experiments of our SSTM implementation shows that it accepts much more workload. Our study has only focused on single-version and another direction would be to investigate multi-versions. As an example, this technique, well-known in the database community [2], can extend VWIR design to accept class $\mathcal{C}3$. Our conclusion is that accepting various workloads requires complex TM mechanisms to test the input and to possibly reschedule it before outputting a consistent history. As an example, SSTM presents a high input acceptance at the cost of using more memory. We expect these results to encourage further research on the best tradeoff between design simplicity and high commit-abort ratio.

Acknowledgments

This work is partially supported by the Velox Project ICT-216852 and the Swiss National Foundation Grant 200021-118043.

References

[1] Utku Aydonat and Tarek S. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08: 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, volume 4167 of *LNCS*, pages 194–208. Springer, 2006.

[4] Pascal Felber, Torvald Riegel, and Christof Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246. ACM, 2008.

[5] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.

[6] Vincent Gramoli, Derin Harmanci, and Pascal Felber. Toward a theory of input acceptance for transactional memories. In *OPODIS '08: Proceedings*

of the 12th International Conference on Principles of Distributed Systems, volume 5401 of *LNCS*, pages 527–533, 2008.

[7] Rachid Guerraoui, Thomas Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *DISC '08: Proceedings of the 22nd International Symposium on Distributed Computing*, volume 5218 of *LNCS*, pages 305–319, 2008.

[8] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*, volume 3724 of *LNCS*, pages 303–323. Springer, 2005.

[9] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184. ACM, 2008.

[10] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[12] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] Jeff Napper and Lorenzo Alvisi. Lock-free serializable transactions. Technical Report TR-05-04, Department of Computer Sciences, University of Texas at Austin, 2005.

[14] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[15] Cristian Perfumo, Nehir Sonmez, Osman Unsal, Adrian Cristal, Mateo Valero, and Tim Harris. Dissecting transactional executions in haskell. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[16] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. From causal to z-linearizable transactional memory. In *PODC '07: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 340–341, New York, NY, USA, 2007. ACM.

[17] Mihalis Yannakakis. Serializability by locking. *J. ACM*, 31(2):227–244, 1984.