Toward a Theory of Input Acceptance for Transactional Memories

Vincent Gramoli[†][‡][*] Derin Harmanci[‡]
Pascal Felber[‡]

[†] EPFL LPD, CH-1015, Switzerland
[‡] University of Neuchâtel, CH-2009, Switzerland

Technical Report LPD-REPORT-2008-009

Abstract

As opposed to database transactional systems, transactional memory (TM) systems are constrained by real-time while treating their input workload. Nevertheless, there is no clear formalization of how a TM should react regarding to a specific input. While TM performance is often measured in terms of throughput, i.e., commit-rate (by time unit), we consider the *commit-abort ratio* of a TM for a given input, as the number of transactions this TM commits over the total number of input transactions. Building onto this, this paper defines the input acceptance of TMs depending on their commit-abort ratio on input classes. To this end, we exhibit several classes of workloads and identify bounds on the input acceptance of existing TM designs. Additionally, we propose a serializable STM that presents high input acceptance at the cost of a more complex algorithm than existing STMs. Finally, experimental validation compares the presented TM designs in terms of input acceptance with realistic workloads.

Keywords. Transactional Memory, Workload, Commit-abort ratio, Input class, Acceptance.

[*]Corresponding author. Address: EPFL-IC-LPD, Station 14, CH-1015, Lausanne, Switzerland. Phone: +41 21 693 8125.

1 Introduction

The role of a transactional system is to receive as an input a stream of events called a workload, to reschedule it with respect to several constraints, and to output a consistent history. In traditional database systems transactional events can be buffered on the server-side before treatment and the response to the client can be delayed. In contrast, Transactional Memory (TM) dedicated to multi-core architectures requires numerous events to be treated upon reception. In fact, the transactional code executed by a core is a stream of events whose interruption would waste cycles. In this paper, we formalize the notion of TM workload into classes of input patterns, whose acceptance helps understanding the performance of a given TM.

TMs are often evaluated in terms of latency and number of commits by time unit (a.k.a. *throughput*). The performance limitation induced by aborted transactions has, however, been neglected. TM optimistically executes a transaction and commits it if no conflict has been detected during its execution. If there exists any risk that a transaction violates consistency, then the transaction is aborted (and its changes are rolled-back) before being restarted. Not only, restarting a transaction raises the probability of conflicts by increasing the total number of executed operations, but the time spent running an aborting transaction is wasted. Hence, lowering the proportion of aborts over commits may increase performance.

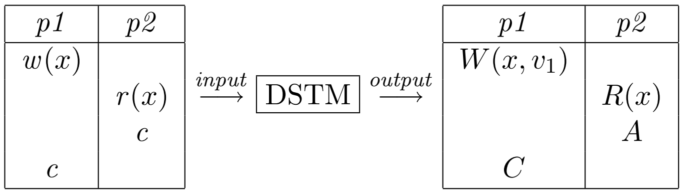


Figure 1: A simple input pattern for which DSTM produces a commit-abort ratio of $\tau = 0.5$ (transaction of *p2* aborts in any case).

Interestingly, many existing TMs unnecessarily abort transactions that could commit without violating consistency. For example, consider the input pattern depicted on the left-hand side of Figure 1, whose pattern events are ordered from top to bottom. DSTM [9], a well-known Software Transactional Memory (STM), would output the series of events on the right-hand side, aborting the transaction executed by thread (or processor) *p2*.[1]

[1]For the sake of simplicity, we assume in this paper a simple contention manager that

Clearly, the read operation applied to variable x could indifferently return value v_1 or the value overwritten without violating serializability [12], opacity [7], or even linerarizability [10]. Nevertheless, since DSTM would assume that the transaction of *p1* owns object x until its commit-time, the read operation of *p2* transaction detects a conflict and aborts.

There are two important observations. First, this pattern appears in many realistic workloads and classical TM benchmarks (e.g., when a transaction deletes an element of a linked list). Second, the proportion of committing transactions among all attempted transactions may degrade significantly for some workloads (e.g., two transactions may repeatedly conflict with each other).

Contributions. This paper defines the notion of commit-abort ratio as the number of committing transactions over the total number of transactions that have completed (either aborted or committed). More importantly, this paper characterizes the workload by presenting inputs and classifying them. This allows us to bound the input acceptance of five existing TM designs:

1. Visible read (VWVR): this design used for instance by SXM [6] let the other threads know of a read operation immediately after the corresponding read request is received;

2. Visible write (VWIR): this design used for instance by DSTM [9] and TinySTM [5] 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 [8] and by TL2 [3] delays the effect of a write operation until reception of the commit request of the same transaction;

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 following constraint: 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.

aborts the offending transaction upon conflict. More sophisticated strategies can help in specific scenarios, e.g., by having a thread wait upon conflict, but in the general case our results do not depend on the contention manager being used.

Finally, we propose a Serializable STM (SSTM) that implements the last design and presents a higher input acceptance than other STMs at the cost of heavy mechanisms to reschedule the input. We validate our results by comparing experimentally the commit-abort ratio of our designs.

Related work. The question whether a set of input transactions can be accepted without being rescheduled has already been studied by Yannakakis [14]. 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 when using locks, and does not take into account any real-time constraint. In contrast here, we especially concentrate on TMs where some operation requests must be treated immediately for efficiency reasons.

Some software transactional memories (STMs) present desirable features that we also target in this paper. Among these STMs, CS-STM permits weaker consistency for the output than serializability in order to accept a wider set of workloads, including long read-only transactions [13], in contrast, we consider only serializable STMs. Other existing STMs are serializable [1, 11] but it is unclear that they do not achieve stronger consistency criteria than serializability. The STM of [11] uses global parameters that may suffer from congestion with many cores, while the input acceptance of TSTM [1] can be improved, as shown here.

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 compares the input classes and Section 5 validates this generalization experimentally. Section 6 discusses the results and concludes.

2 Model and Definitions

The role of a TM is to receive a workload and to produce efficiently an associated history that satisfies consistency. This section formalizes the notions of workload and history as TM input and TM output, respectively. In this model, we assume that a transaction abort implies to retry the transaction later—the retried transaction is then considered as a distinct one.

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. 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, respectively. In the definition of TM input, accessed variables are of particular interest since they order operations of distinct transactions. However, the values read and written are of no interest here 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 pair $\langle I, \prec \rangle$ where I is a set of input events and $\prec$ is a total order defined over I. The relation $\prec$ corresponds intuitively to the real-time order, 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 variables will be ordered by the TM (e.g., using a compare-and-swap) and non-conflicting concurrent events can be arbitrarily ordered. An *input class* $\mathcal{C}$ can be an infinite set of input patterns. 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 event are included).

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 complete read or write operation, a commit, or an abort. We refer to the complete 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 complete 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 complete read operation or a complete 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 order 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. 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 commit or abort. By abuse of notation, we refer indifferently to a transaction as an input transaction or its associated output transaction.

Consistency. The consistency criterion considered in this paper is serializability. More precisely, the TMs presented here output histories that satisfy conflict-serializability [4].

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 have been removed.

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 $\pi_1 \longrightarrow \pi_2$. 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 *serializable* if and only if the serializability graph of its complete history $C(H)$ is acyclic [2,4]. By extension, a TM is serializable if and only if it outputs only serializable histories.

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 regular 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 negation notation, '$\neg$', indicates that any event of the following set is prohibited. Finally, the choice notation, '$|$', denotes the occurrence of either the preceding or the following set of events. Operators are ordered by priority as $\neg, *, |$.

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 On the Input Acceptance of TM Designs

This section identifies several TM designs and upper-bounds their input acceptance. 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. We will discuss multi-version in Section 6.

TM constraints. When processing a given operation of the input, the TM has to take an appropriate action. Upon read, the TM has to return some value (which might be the latest committed version, an older version in a multi-version TM, or a yet-uncommitted version). Similarly, upon write, the TM can make the written value visible to other threads or delay the publication of the value until some point in the future (typically commit-time). These broad decisions define designs of TM.

TM designs. 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).

We identify some TM designs and we exhibit their performance bounds by defining input classes that they do not accept. To illustrate our discussion, we use five different TM designs: *(i)* a design with visible reads and visible writes similar to SXM [6]; *(ii)* a design with visible writes and invisible reads that resembles DSTM [9] and TinySTM [5]; *(iii)* a design with invisible (delayed) writes and invisible reads similar to WSTM [9] and TL2 [3]; *(iv)* a design alike the recently proposed TSTM [1] that allows serialization points to precede commit-time; *(v)* a design that keeps track of all variables involved in a possible conflict (we propose a new STM that implements this design).

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). While we do not claim that these classes are maximal we claim that their patterns are realistic and express important design limitations in terms of commit-abort ratio. 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 [9] or in-order lock acquisition [8]. 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 [6]. For simplicity of presentation, we assume that variables are versioned.

Algorithm 1 VWVR Design

```
 1: State of transaction $t$:
 2:    $\textit{read-set} \subset X$, initially $\emptyset$
 3:    $\textit{write-set} \subset X \times V$, initially $\emptyset$

 4: State of variable $x$:
 5:    $\textit{val} \in V$, initially default value
 6:    $\textit{writer} \in T$, initially $\perp$
 7:    $\textit{readers} \subset T$, initially $\emptyset$

 8: read$(x)_t$:
 9:    if $\langle x, v' \rangle \in \textit{write-set}$ then $v \leftarrow v'$
10:    else
11:       if $x.\textit{writer} \neq \perp$ then abort()
12:       $v \leftarrow$ last committed value of $x$
13:       $\textit{read-set} \leftarrow \textit{read-set} \cup \{x\}$
14:       $x.\textit{readers} \leftarrow x.\textit{readers} \cup \{t\}$
15:    return $v$

16: write$(x, v)_t$:
17:    if $x.\textit{readers} \setminus \{t\} \neq \emptyset$ then abort()
18:    if $x.\textit{writer} = t$ then
19:       $\textit{write-set} \leftarrow \textit{write-set} \setminus \{\langle x, * \rangle\} \cup \{\langle x, v \rangle\}$
20:    else
21:       if $x.\textit{writer} \neq \perp$ then abort()
22:       $\textit{write-set} \leftarrow \textit{write-set} \cup \{\langle x, v \rangle\}$
23:       $x.\textit{writer} \leftarrow t$

24: commit$()_t$:
25:    for each $\langle x, v \rangle \in \textit{write-set}$ do
26:       $x.\textit{val} \leftarrow v$
27:       $x.\textit{writer} \leftarrow \perp$
28:    for each $\langle x \rangle \in \textit{read-set}$ do
29:       $x.\textit{readers} \leftarrow x.\textit{readers} \setminus \{t\}$

30: abort$()_t$:
31:    for each $\langle x, v \rangle \in \textit{write-set}$ do
32:       $x.\textit{writer} \leftarrow \perp$
33:    for each $\langle x \rangle \in \textit{read-set}$ do
34:       $x.\textit{readers} \leftarrow x.\textit{readers} \setminus \{t\}$
```

If a read request is input, the TM records the transaction in $x.\textit{readers}$ (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 $x.\textit{writer}$ (Line 23).

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

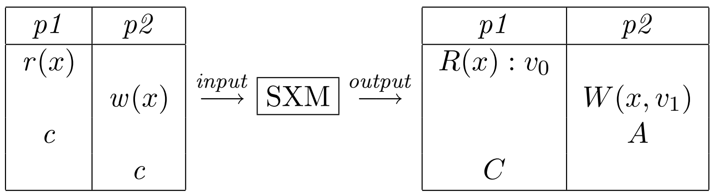


Figure 2: An input pattern for which SXM produces a commit-abort ratio of $\tau = 0.5$ (transaction of *p2* aborts upon writing).

Theorem 1 *There is no TM implementing VWVR design that accepts any input pattern of the following class:*

$$\mathcal{C}1 = \pi^*(\pi_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* \pi_i^x)\pi^*.$$

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 any additional event added to $\mathcal{P}1$ or $\mathcal{P}1'$ that results in a pattern of $\mathcal{C}1$ 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

This section introduces a TM design with visible writes and invisible reads, called *VWIR*, that is similar to DSTM [9] with a contention manager that aborts the transaction detecting a conflit. 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 the same functions as in the previous Algorithm except that we 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 21).

Algorithm 2 VWIR Design

```
1: State of transaction $t$:
2:   $read\text{-}set \subset X$, initially $\emptyset$
3:   $write\text{-}set \subset X \times V$, initially $\emptyset$

4: State of variable $x$:
5:   $val \in V$, initially default value
6:   $writer \in T$, initially $\bot$

7: read$(x)_t$:
8:   if $\langle x, v' \rangle \in write\text{-}set$ then $v \leftarrow v'$
9:   else
10:    if $x.writer \neq \bot$ then abort()
11:    validate()
12:    $v \leftarrow$ last committed value of $x$
13:    $read\text{-}set \leftarrow read\text{-}set \cup \{x\}$
14:  return $v$

15: write$(x, v)_t$:
16:  if $x.writer = t$ then
17:    $write\text{-}set \leftarrow write\text{-}set \setminus \{\langle x, * \rangle\} \cup \{\langle x, v \rangle\}$
18:  else
19:    if $x.writer \neq \bot$ then abort()
20:    $write\text{-}set \leftarrow write\text{-}set \cup \{\langle x, v \rangle\}$
21:    $x.writer \leftarrow t$

22: commit$()_t$:
23:  validate()
24:  for each $\langle x, v \rangle \in write\text{-}set$ do
25:    $x.val \leftarrow v$
26:    $x.writer \leftarrow \bot$

27: abort$()_t$:
28:  for each $\langle x, v \rangle \in write\text{-}set$ do
29:    $x.writer \leftarrow \bot$

30: validate$()_t$:
31:  for each $x \in read\text{-}set$ do
32:    $x' \leftarrow$ last committed version of $x$
33:    if $x \neq x'$ then abort()
```

Common input patterns are neither accepted by this design. Consider the input pattern depicted in Figure 1 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 aborting a transaction: while in this design the transaction t_2 aborts due to this conflict, any alternative contention manager aborts one of the current transactions.[2] As stated in the following theorem, an input class including this pattern is not accepted by this design.

Theorem 2 *There is no TM implementing VWIR design that accepts any input pattern of the following class:*

$$\mathcal{C}2 = \pi^*(r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \mid w_j^x \neg c_j^* r_i^x)\pi^*.$$

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

Proof. The proof is similar to the proof of Theorem 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 any additional event added to $\mathcal{P}2$ or $\mathcal{P}2'$ that results in a pattern of $\mathcal{C}2$ is not accepted by VWIR STMs for the same reason as above. $\square$

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

3.3 IWIR Design

Here, we propose a second design that accepts patterns of the preceding class, i.e., for which the previous impossibility result does not hold. We do not claim that all patterns of $\mathcal{C}1$ are accepted by this design. This design, inspired by WSTM [8], 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 is different from previous designs due to the invisibility of writes: while modifications are recorded at write-time in the *write-set*, these modifications are made visible no sooner than at commit-time. The corresponding functions and states are presented in Algorithm 3.

Even IWIR design does not accept some very common input patterns. As illustrated in Figure 3, assume that a transaction t_2 writes a variable x and commits after another transaction t_1 reads x but before t_1 commits. Because t_1 has read the previous value, it fails its validation at commit-time and aborts. This is a classical example of a transaction reading a value that is later overwritten by another transaction. Such a pattern also arises when performing concurrent operations on a linked list. The following theorem gives a set of input patterns that are not accepted by STMs of the IWIR design.

Theorem 3 *There is no TM implementing IWIR design that accepts any*

Algorithm 3 IWIR Design

```
State of transaction $t$:
    $read\text{-}set \subset X$, initially $\emptyset$
    $write\text{-}set \subset X \times V$, initially $\emptyset$

State of variable $x$:
    $val \in V$, initially default value

read$(x)_t$:
    if $\langle x, v' \rangle \in write\text{-}set$ then $v \leftarrow v'$
    else
        validate()
        $v \leftarrow$ last committed value of $x$
        $read\text{-}set \leftarrow read\text{-}set \cup \{x\}$
    return $v$

write$(x, v)_t$:
    $write\text{-}set \leftarrow write\text{-}set \setminus \{\langle x, * \rangle\} \cup \{\langle x, v \rangle\}$

commit$()_t$:
    validate()
    for each $x \in write\text{-}set$ do
        $x' \leftarrow$ last committed version of $x$
        if $x \neq x'$ then abort()
    for each $\langle x, v \rangle \in write\text{-}set$ do
        $x.val \leftarrow v$

abort$()_t$: —

validate$()_t$:
    for each $x \in read\text{-}set$ do
        $x' \leftarrow$ last committed version of $x$
        if $x \neq x'$ then abort()
```

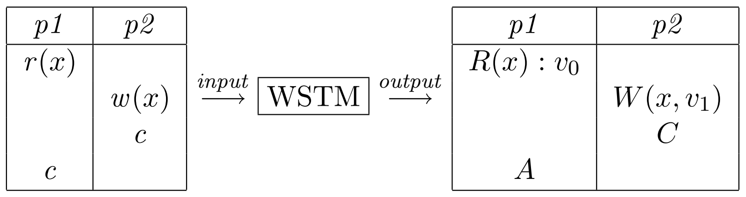


Figure 3: A simple input pattern for which STMs implementing IWIR design produce a commit-abort ratio of $\tau = 0.5$ (transaction of *p2* aborts in any case).

input pattern of the following class:

$$\mathcal{C}3 = \pi^*(r_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* r_i^x) \neg c_i^* c_j \pi^*.$$

Proof. The proof relies on the existence of two minimal patterns belonging to $\mathcal{C}3$ that IWIR STMs never accept. We show, by contradiction, that each of these patterns is not accepted.

First, consider the following input pattern: $\mathcal{P}3 = r(x)_i w(x)_j c_j$ ($\mathcal{P}3 \in \mathcal{C}3$), 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 following input pattern: $\mathcal{P}3' = w(x)_j r(x)_i c_j$ ($\mathcal{P}3' \in \mathcal{C}3$), 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 and upon committing, t_i fails in validating leading to an abort.

Both patterns lead to an abort that contradicts the assumption. 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 class $\mathcal{C}1$, since $\mathcal{C}1$ is a subset of $\mathcal{C}3$ as we indicate in Section 4.

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 input. In this sense, this design relaxes the commit time and we call it *Commit-Time Relaxation (CTR)*. More precisely, it allows to advance the commit time of transactions. 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 read(x) request forces the clock of the transaction to be at least as large as the clock of the last transactions 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 yet completed, an already committed transaction t_1 may force t_2 to be serialized before.

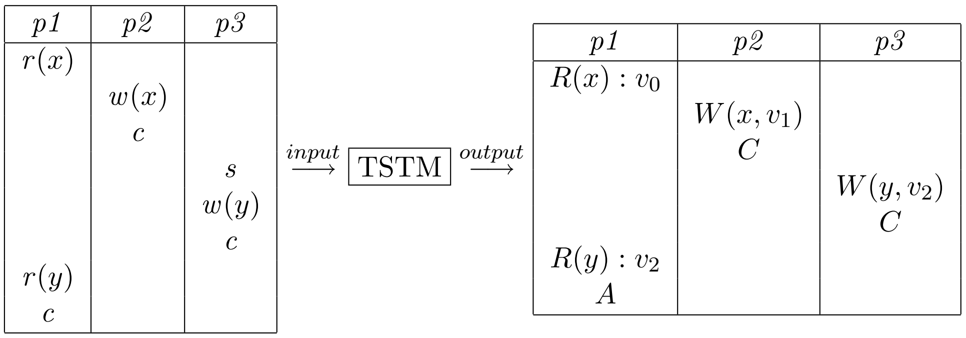
TSTM is claimed to achieve conflict-serializability, however, it might not accept all possible conflict-serializations. Figure 4 presents an input pattern that TSTM does not accept: since transactions choose their clock depending on the last committed version of the object they access, two transactions may choose the same clock and forces another to abort. In this example, transactions of $p2$ and $p3$ chooses the same clock and force $p1$ transaction to abort.

This pattern typically happens when a long transaction t runs concurrently with short update transactions that update the variables read by t. Note that, if the update transactions do not conflict, they do not need to be

Algorithm 4 CTR Design

```
State of transaction $t$:
    $status \in \{\mathsf{active}, \mathsf{inactive}\}$, initially $\mathsf{active}$
    $read\text{-}set \subset X$, initially $\emptyset$
    $write\text{-}set \subset X \times V$, initially $\emptyset$
    $clock\text{-}int$, a record with fields:
        $lb \in \mathbb{N}$, initially 0
        $ub \in \mathbb{N}$, initially $\infty$
    $clock \in \mathbb{N} \cup \{\bot\}$, initially $\bot$
    $n \in \mathbb{N}$, the number of threads

State of variable $x$:
    $val \in V$
    $clock \in \mathbb{N}$, initially 0
    $active\text{-}readers \subset T$, initially $\emptyset$

read$(x)_t$:
    $x.active\text{-}readers \leftarrow x.active\text{-}readers \cup \{t\}$
    $clock\text{-}int.lb \leftarrow \max(x.clock, clock\text{-}int.lb)$
    if $clock\text{-}int.ub < clock\text{-}int.lb$ then $\mathsf{abort}()$
    $read\text{-}set \leftarrow read\text{-}set \cup \{x\}$
    return $x$

write$(x, v)_t$:
    $write\text{-}set \leftarrow write\text{-}set \setminus \{\langle x, *\rangle\} \cup \{\langle x, v\rangle\}$

commit$()_t$:
    for any $\langle x, *\rangle \in write\text{-}set$ do
        $clock\text{-}int.lb \leftarrow \max(x.clock, clock\text{-}int.lb)$
        if $clock\text{-}int.ub \neq \infty$ then
            $clock \leftarrow clock\text{-}int.ub$
            if $clock < clock\text{-}int.lb$ then $abort()$
        else
            $clock \leftarrow clock\text{-}int.lb + n$
            if $clock > clock\text{-}int.ub$ then $abort()$
        for any $r \in x.active\text{-}readers$ do
            if $r.status \neq \mathsf{active}$ then
                $x.active\text{-}readers \leftarrow x.active\text{-}readers \setminus \{r\}$
            else
                $r.clock\text{-}int.ub \leftarrow clock - 1$
    for $\langle x, v\rangle \in write\text{-}set$ do
        $x.clock \leftarrow clock$
        $x.val \leftarrow v$
    $status \leftarrow \mathsf{inactive}$

abort$()_t$:
    $status \leftarrow \mathsf{inactive}$
```

ordered according to real-time order. The following theorem generalizes this result by showing that STMs implementing CTR design does not accept a new input class.



p1	*p2*	*p3*
$r(x)$		
	$w(x)$	
	c	
		s
		$w(y)$
		c
$r(y)$		
c		

$\xrightarrow{input} \boxed{\text{TSTM}} \xrightarrow{output}$

p1	*p2*	*p3*
$R(x) : v_0$		
	$W(x, v_1)$	
	C	
		$W(y, v_2)$
		C
$R(y) : v_2$		
A		

Figure 4: An input pattern that TSTM does not accept. The commit-abort ratio obtained for TSTM is $\tau = \frac{2}{3}$ (transactions of *p2* and *p3* commit but transaction of *p1* aborts).

Theorem 4 *There is no TM implementing CTR design that accepts any input pattern of the following class:*

$$\mathcal{C}4 = (\neg w^x)^* r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \neg c_i^* s_k \neg (c_i \,|c_k \mid r_k^x)^* w_k^y \neg (c_i \,|c_k \mid r_k^x)^* c_k \neg c_i^* r_i^y \pi^*.$$

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 CTR-STM. 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 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 to n too, when committing. Consequently, t_i has a larger lower-bound n than its upper-bound $n-1$, that is, t_i aborts.

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, 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 [7] or linearizable [10]). 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 3, 2, 1, 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 [1,11] that may experience congestion when scaling to large numbers of cores. Figure 5 presents the pseudocode of SSTM. As mentioned earlier and like previous designs, functions are assumed to execute atomically for the sake of simplicity in the presentation.

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 $t.\textit{past-tx}$. Transaction t detects those transactions either because they are in $x.\textit{active-readers}$ (Line 38) or precede one of these (Line 37), or because they are in $x.\textit{write-fc}$ (Lines 23 and 38) or precede one of these (Lines 22 and 37). Transaction t also keeps track of its succeeding transactions in $t.\textit{future-tx}$ 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 $t'.\textit{past-tx}$ and $t'.\textit{future-tx}$. Transaction t may abort for two reasons. First, if it appears to precede itself in the conflict graph (Lines 21 and 36). Second, if there exists a transaction that t precedes but that also precedes t (Lines 26 and 41). 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

Algorithm 5 SSTM (Serializable STM) - Part 1

```
1: State of transaction $t$:
2:    $status \in \{\mathsf{active}, \mathsf{inactive}\}$, initially active
3:    $write\text{-}set \subset X \times V$, initially $\emptyset$
4:    $read\text{-}set \subset X$, initially $\emptyset$
5:    $past\text{-}tx \subset T$, initially $\emptyset$ // the previous tx in the conflict graph
6:    $future\text{-}tx \subset T$, initially $\emptyset$ // the next tx in the conflict graph

7: State of shared variable $x$:
8:    $write\text{-}fc \subset T$, initially $\emptyset$ // the write future conflicts
9:    $active\text{-}readers \subset T$, initially $\emptyset$ // the active reader tx
10:   $val \in V$, initially the default value

11: write$(x, v)_t$:
12:   $write\text{-}set \leftarrow (write\text{-}set \setminus \{\langle x, * \rangle\}) \cup \{\langle x, v \rangle\}$

13: read$(x)_t$:
14:   $read\text{-}set \leftarrow read\text{-}set \cup \{x\}$
15:   if $\langle x, v' \rangle \in write\text{-}set$ then
16:      $v \leftarrow v'$
17:   else
18:      $x.active\text{-}readers \leftarrow x.active\text{-}readers \cup \{t\}$
19:      for all $t'$ in $x.write\text{-}fc$ do
20:         for all $t'' \in t'.past\text{-}tx$ do
21:            if $t = t''$ then abort()
22:            $past\text{-}tx \leftarrow past\text{-}tx \cup \{t''\}$
23:         $past\text{-}tx \leftarrow past\text{-}tx \cup \{t'\}$
24:      for all $t'$ in $past\text{-}tx$ do
25:         for all $t'' \in future\text{-}tx$ do
26:            if $t' = t''$ then abort()
27:            $t'.future\text{-}tx \leftarrow t'.future\text{-}tx \cup \{t''\}$
28:         $t'.future\text{-}tx \leftarrow t'.future\text{-}tx \cup \{t\}$
29:      $v \leftarrow x.val$
30:   return $v$
```

graph later.

Tracking all conflicts is known to be a difficult task [7] while it is easy to check linearizability in a composed manner [10], 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

Algorithm 6 SSTM - Part 2

```
31: commit()$_t$:
32:   for all $\langle x, v\rangle \in$ write-set do
33:     $x$.write-fc $\leftarrow x$.write-fc $\cup \{t\}$
34:     for all $t' \in x$.active-readers $\cup\ x$.write-fc do
35:       for all $t'' \in t'$.past-tx do
36:         if $t = t''$ then abort()
37:         past-tx $\leftarrow$ past-tx $\cup \{t''\}$
38:       if $t \neq t'$ then past-tx $\leftarrow$ past-tx $\cup \{t'\}$
39:     for all $t'$ in past-tx do
40:       for all $t'' \in$ future-tx do
41:         if $t' = t''$ then abort()
42:         $t'$.future-tx $\leftarrow t'$.future-tx $\cup \{t''\}$
43:       $t'$.future-tx $\leftarrow t'$.future-tx $\cup \{t\}$
44:   for all $\langle x, v\rangle \in$ write-set do
45:     $x$.val $\leftarrow v$
46:   status $\leftarrow$ inactive
47:   c-clean()

48: abort()$_t$:
49:   status $\leftarrow$ inactive
50:   a-clean()

51: a-clean()$_t$:
52:   for all $x$ such that $\langle x, *\rangle \in$ write-set or $x \in$ read-set do
53:     $x$.write-fc $\leftarrow x$.write-fc $\setminus \{t\}$
54:     $x$.active-readers $\leftarrow x$.active-readers $\setminus \{t\}$
55:   for all $t' \in$ past-tx do
56:     $t'$.future-tx $\leftarrow t'$.future-tx $\setminus \{t\}$
57:   for all $t' \in$ future-tx do
58:     $t'$.past-tx $\leftarrow t'$.past-tx $\setminus \{t\}$
59:   free($t$)

60: c-clean()$_t$:
61:   for all $x$ such that $\langle x, *\rangle \in$ read-set do
62:     $x$.active-readers $\leftarrow x$.active-readers $\setminus \{t\}$
63:   for all $t' \in T$ do
64:     if $t'$.status = inactive and $t'$.past-tx $= \emptyset$ then
65:       past-tx $\leftarrow$ past-tx $\setminus \{t'\}$
66:       for all $t'' \in t'$.future-tx do
67:         $t''$.past-tx $\leftarrow t''$.past-tx $\setminus \{t'\}$
68:       for all $x$ such that $\langle x, *\rangle \in t'$.write-set do
69:         $x$.write-fc $\leftarrow x$.write-fc $\setminus \{t'\}$
70:       free($t'$)
```

TSTM, in Figure 4 (center and left-hand side).

Theorem 5 *SSTM is not constrained by real-time order.*

Proof. A simple counter-example is presented in Figure 5. Clearly, the

p1	*p2*	*p3*
$r(x)$		
	$w(x)$	
	c	
		s
		$w(y)$
		c
$r(y)$		
c		

$\xrightarrow{input} \boxed{\text{SSTM}} \xrightarrow{output}$

p1	*p2*	*p3*
$R(x) : v_0$		
	$W(x, v_1)$	
	C	
		$W(y, v_2)$
		C
$R(y) : v_2$		
C		

Figure 5: An input pattern that SSTM accepts while TSTM does not accept it. (The commit-abort ratio obtained for SSTM is 1.)

input (right) is accepted by SSTM resulting in the output on the left-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 order 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. □

SSTM presents promising input acceptance compared to numerous existing STMs. The drawback is the high cost of rescheduling the input into a serializable output. In fact, this is due to the large number of shared variables that must be recorded as being part of a potential conflict to identify cycles in the serialization graph. In terms of memory space this cost is upper-bounded by the maximal number of shared variables and by the maximal number of concurrent transactions. That is, the memory cost is proportional to the minimum of these two factors.

3.6 Correctness Proof

Here we show that SSTM, presented above, is conflict-serializable.

Lemma 6 *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 $x.\mathit{write\text{-}fc}$ at Line 33. 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 38 and symmetrically adds itself in t_0.*future-tx* at Line 43.

Second if t_0 does not write but reads x before t_1 writes x, then t_0 adds itself to $x.\mathit{active\text{-}reader}$ at Line 18 so that t_1 adds it to t_1.*past-tx* at Line 38. Again symmetrically, t_1 inserts itself into t_0.*future-tx* at Line 43. 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 7 *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 $\tau' < \tau$, $t_2 \in t_1$.*future-tx* and $t_1 \in t_2$.*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 38) and itself to t_0.*future-tx*at Line 28 (resp. at Line 43) but also t_2 at Line 27 (resp. at Line 42), which belongs to its t_1.*future-tx*, to t_0.*future-tx*.

In case $\tau < \tau'$, $t_0 \in t_1$.*past-tx* at time τ'. Assume t_2 conflicting operation is a read (resp. a write). Transactions t_0, which belongs to t_1.*past-tx*, and t_1 are inserted in t_2.*past-tx* at Line 23 (resp. at Line 38), 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 43). □

Lemma 8 *$t \notin t$.future-tx.*

Proof. Assume that $t \in t$.*future-tx* holds, we proceed by contradiction. Transaction t can only be inserted in t.*future-tx* at Line 28 or at Line 43 because neither reaching Line 27 nor Line 42 with $t = t'$ is possible as transaction t would abort prior to that (Lines 26 and 41). As a result, t was already in t.*past-tx* when Line 28 or 43 has been reached.

Now we show that t cannot be inserted in t.*past-tx* leading to the contradiction. If t already belongs $t \in x$.*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.*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.*past-tx* because $t = t'$ (Line 38). $\square$

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 9 *In all histories H of SSTM, there is no path $p = t_1 \longrightarrow \ldots \longrightarrow t_k \longrightarrow t_1$ 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 6 we know that $p = t_1 \longrightarrow \ldots \longrightarrow t_k \longrightarrow t_1$ implies that $t_{i+1} \in t_i$.*future-tx* for all i such that $0 < i \leq k-1$ and $t_1 \in t_k$.*future-tx*. Second, by the transitivity property of Lemma 7 we obtain that $t_i \in t_i$.*future-tx* ($0 < i \leq k$) which contradicts Lemma 8. $\square$

Theorem 10 *SSTM is conflict-serializable.*

Proof. The proof follows from the conjunction of Corollary 9 and Theorem 2.1 of [2]. $\square$

4 Class Comparison

The previous section 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 $\mathcal{C}$.

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 class $\mathcal{C}0$, $\mathcal{C}1$, and $\mathcal{C}2$. For instance, as stated in Theorem 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

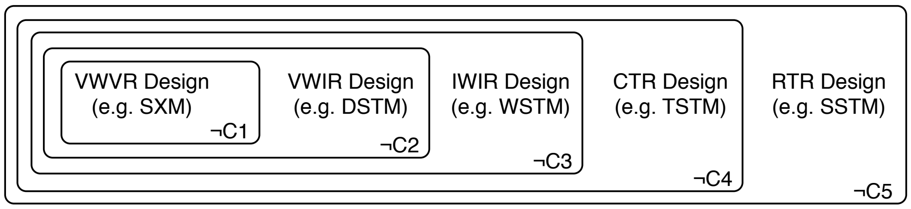


Figure 6: Hierarchization of classes. The VWVR design accepts no input patterns of the presented classes, the VWIR design accepts inputs that are not in classes ranging from $\mathcal{C}2$ to $\mathcal{C}4$, the IWIR design accepts inputs that are neither in $\mathcal{C}3$ nor in $\mathcal{C}4$, the CTR design accepts input patterns only outside $\mathcal{C}4$. Finally, we have not yet identified patterns not accepted by design RTR.

represent that a TM accepts patterns that are out of 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. The hierarchy shown in Figure 6 compares the input acceptance of these TM designs.

5 Experimental Validation

We have implemented the VWIR, IWIR, CTR and RTR designs and run some experiments on an 8-core Intel Xeon machine. We have chosen the following sorted linked list benchmark to contrast the performance of those designs: Initially, the benchmark inserts 256 elements in the linked list. Then, each thread starts and executes either a *contains* or an *update* transaction. We call the probability of executing an update transaction *update probability.* Contain transactions look for a value in the linked list and read through all elements in order until it finds the searched value. The update transactions alternatively insert a new element or delete the last inserted element, thus maintaining the size of the linked list roughly constant during the experiment. Since the linked list is sorted, update transactions will also read the elements of the linked list in sequence, until it finds the location where to insert/delete the element to be updated. Thus, update transactions perform write operations at the end of the transaction right before commit.

The first type of experiments we have performed is to contrast the acceptance of the designs under different workload conditions and to validate the hierarchy presented in Figure 6. For this experiment, the sorted linked list benchmark is run with all the different designs and we recorded the average commit-abort ratio of each design for different update probabilities. The results obtained are depicted in Figure 7.

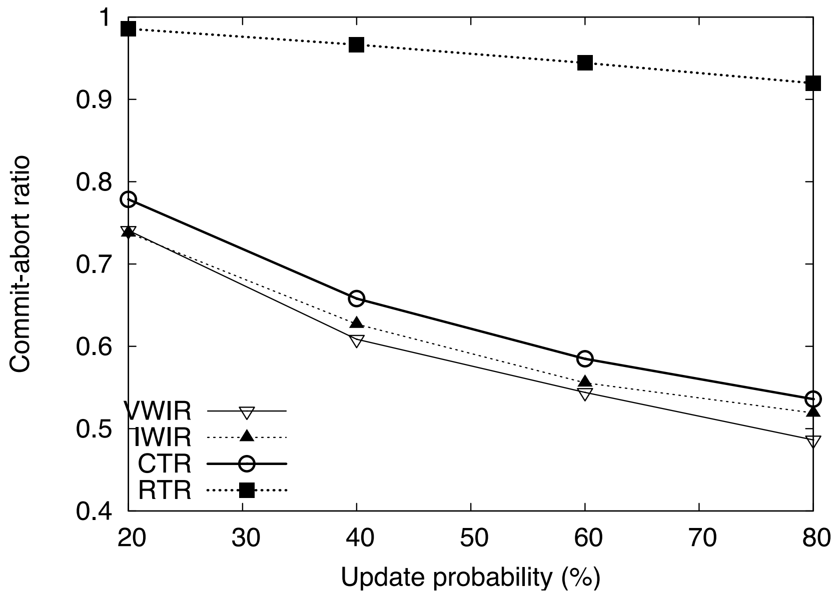


Figure 7: Analysis of acceptance under different workload conditions. Comparison of average commit-abort ratio of VWIR, IWIR, CTR and RTR designs, on a 256 element linked list as the update probability varies.

The figure shows, for the RTR, CTR and IWIR classes, that the higher a design in the hierarchy the higher its commit-abort ratio (thus the higher its acceptance). In other words, the figure shows how classes $\mathcal{C}4$ and $\mathcal{C}3$ distinguish between the acceptances of RTR, CTR and IWIR classes. We observe, on the other hand, that the acceptance of VWIR and IWIR classes is practically the same for all the update probabilities. That is because the sorted linked list benchmark has a very low probability of generating patterns of class $\mathcal{C}2$ which separates those designs in the hierarchy. For a $\mathcal{C}2$

class pattern to occur, a $r(x)$ operation should be performed by a transaction some time after a $w(x)$ operation of an update transaction, which is very seldom since update transactions are committed right after a $w(x)$ operation they perform.

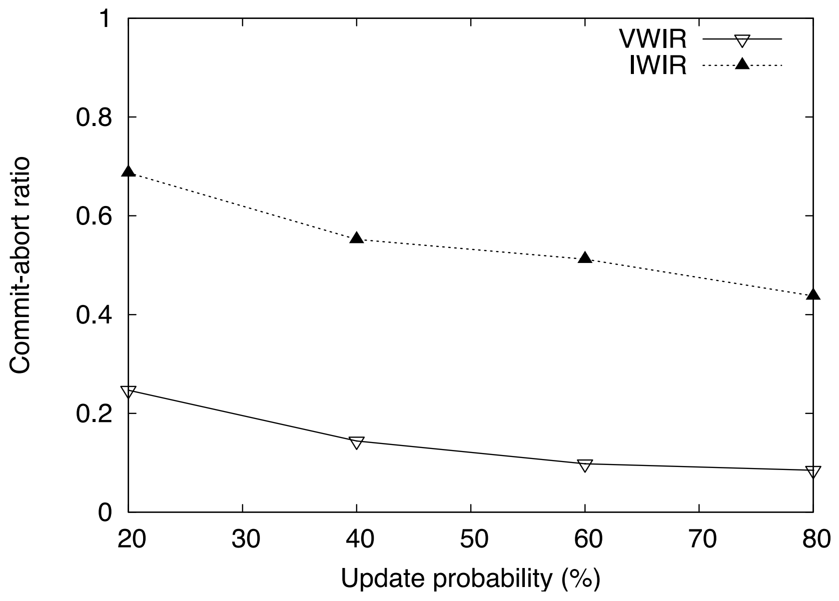


Figure 8: Analysis of acceptance under different workload conditions. Comparison of average commit-abort ratio of VWIR and IWIR, on a 256 element linked list as the number of cores varies.

Since the first experiment does not distinguish between the acceptance of VWIR and IWIR designs, we have performed a second experiment specifically to contrast the acceptance of those two designs. This experiment ensures the occurrence of patterns in class $\mathcal{C}1$ by introducing the following modification to the first experiment: the update transactions after having updated the list looks for a value in the linked list. The results obtained for this second experiment are illustrated in Figure 8. This figure shows that for all the update probabilities the IWIR design has a higher acceptance than the VWIR design. Consequently, the results of this figure together with Figure 7 validates the hierarchy presented in Figure 6.

We have designed a third experiment to analyze the scalability of each design. In this experiment, we use the sorted linked list benchmark used in the first experiment, measure the average commit-abort ratio by varying

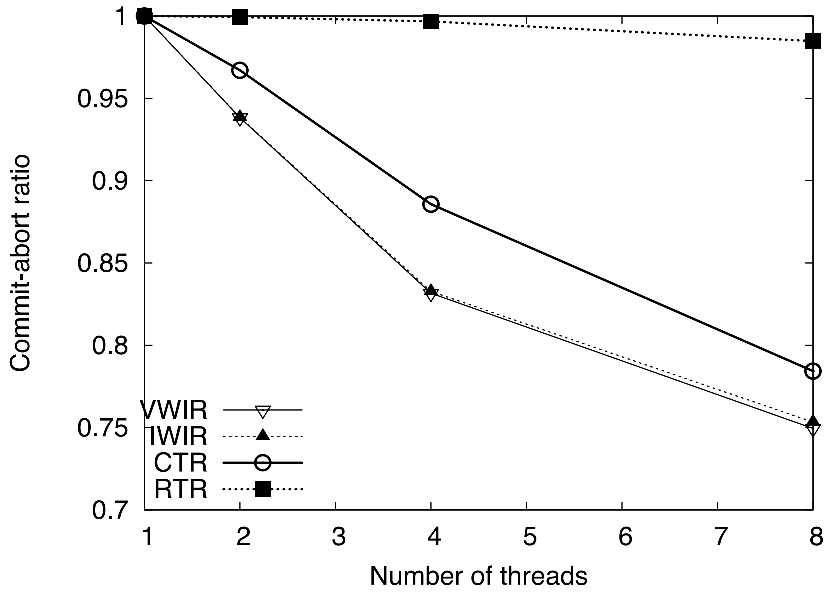


Figure 9: Comparison of average commit-abort ratio scalability of VWIR, IWIR, CTR and RTR designs under 20% update probability workload (linked-list size is again 256).

the number of threads for a fixed update probability of 20%. The results of the experiment are shown in Figure 9. This figure clearly illustrates that the acceptance of RTR design is practically not affected by the increasing number of threads, while the other designs have a decreasing acceptance as the number of threads increase. This result indicates how well RTR design copes with conflicts that span transactions of multiple threads.

6 Discussion and Conclusion

Discussion. The purpose of commit-abort ratio is not to capture the absolute performance of a TM but rather to outline important causes of the lack of performance. As already mentioned, the classes described here are not maximal in the sense that they do not represent all non-accepted patterns. A next step is to identify, for each given design, all the patterns that are not accepted. This identification, together with the hierarchization proposed in this paper, would outline the best design in all circumstances.

Our study has only taken into account designs with a single current version by shared object at any time. A way of extending designs is to tolerate multi-version, i.e., letting multiple versions of the same object cohabit at the same time. As an example, this technique, well-known in the database community, can extend VWIR design to accept class $\mathcal{C}2$.

Conclusion. We defined the input acceptance for transactional memories using a hardware-independent performance metric, called commit-abort ratio. This metric led us to investigate the behavior of TMs in response to classified workloads and to upper-bound the input acceptance of some TM designs proposed in the STM literature. Our conclusion is that accepting various workloads requires important TM overheads to test the input and to possibly reschedule it before outputting a consistent history. This observation is illustrated by a new STM, SSTM, that may exhibit long latency in practical settings, but provides a much larger commit-abort ratio compared to existing designs. We expect this result to encourage further research on the best tradeoff between low latency and high commit-abort ratio.

References

[1] Utku Aydonat and Tarek S. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08: Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*. ACM, 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 Berlin / Heidelberg, 2006.

[4] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[5] 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, February 2008.

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

[7] 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, February 2008.

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

[9] 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 twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

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

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

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

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

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