Toward a Theory of Input Acceptance for Transactional Memories*

Vincent Gramoli[1,2], Derin Harmanci[2], and Pascal Felber[2]

[1] School of Computer and Communication Sciences, EPFL, Switzerland
vincent.gramoli@epfl.ch
[2] University of Neuchâtel, CH-2009, Switzerland
{derin.harmanci,pascal.felber}@unine.ch

1 Introduction

Transactional memory (TM) systems receive as an input a stream of events also known as a *workload*, reschedule it with respect to several constraints, and output a consistent history. In multicore architectures, the transactional code executed by a processor is a stream of events whose interruption would waste processor 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 throughput (number of commits by time unit). The performance limitation induced by aborted transactions has, however, been mostly neglected. A 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 this transaction does not commit. Since stopping a thread until possible conflict resolution would waste core cycles, the common solution is to choose one of the conflicting transactions and to abort it.

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 events are ordered from top to bottom. DSTM [1], a well-known Software Transactional Memory (STM), would detect a conflict and try to resolve it, whatever it costs. Clearly, the read operation applied to variable x could indifferently return value v_1 or the value overwritten without violating serializability [2], or even opacity [3], thus no conflict resolution is needed. This paper focuses on the ability of TMs to commit transactions from given workloads: the input acceptance of TMs.

Contributions. In this paper, we upper-bound the input acceptance of existing TMs by grouping them into the following designs. *(i)* Visible write (VWIR); *(ii)* Invisible write (IWIR); *(iii)* Commit-time relaxation (CTR); *(iv)* Real-time relaxation (RTR). We propose a Serializable STM, namely *SSTM*, that implements the last design in a fully decentralized fashion. Finally, we compare the

* Corresponding author: Vincent Gramoli, EPFL-IC-LPD, Station 14, CH-1015 Lausanne, Switzerland; fax: +41 21 693 7570. This work is supported by the Velox Project and the Swiss National Foundation Grant 200021-118043.

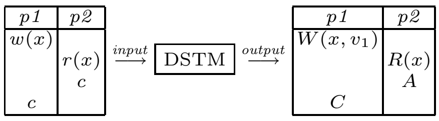

Fig. 1. A simple input pattern for which DSTM produces a commit-abort ratio of $\tau = 0.5$ (e.g., transaction of *p2* aborts with contention manager Polite that kills the transaction detecting the conflict)

four TM designs based on the upper-bound of their input acceptance. We validate our theoretical comparison experimentally under realistic workloads.

Related Work. The question whether a set of input transactions can be accepted without being rescheduled has already been studied by Yannakakis [4]. In contrast here, we especially concentrate on TMs where some operation requests must be treated immediately for efficiency reasons. 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. As far as we know SSTM is, however, the first of these STMs that is fully decentralized and ensure serializability. CS-STM [5] is decentralized but is not serializable. Existing serializable STMs require either centralized parameters [6] or a global reader table [7] to minimize the number of aborting transactions.

2 Model and Definitions

This section formalizes the notions of workload and history as TM input and TM output, respectively. In this model, we assume that all accesses are executed inside a transaction, each thread executes one transaction at a time, and when a transaction aborts it must be retried later—the retried transaction is then considered as a distinct one.

First, we formalize the workload as the TM input that contains a series of events in some transaction t. These *input events* are a start request, s_t, an operation call on variable x, $\pi(x)_t$ (either a read $r(x)_t$ / ${r^x}_t$ or a write $w(x)_t$ / ${w^x}_t$) or a commit request c_t. We refer to an *input pattern* $\mathcal{P}$ of a TM as a sequence of input events. The sequence order 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. 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. 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 $W(x, v_1)_t$. We refer 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. 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 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 $\neg, *, |$.

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). The commit-abort ratio is an important measure of "achievable concurrency" for TM performance. In the remainder of the paper, we say that a TM *accepts* an input pattern if it commits all of its transactions, i.e., $\tau = 1$.

3 On the Input Acceptance of Existing TMs

This section identifies several TM designs and upper-bounds their input acceptance. All the designs considered here use contention manager Polite, i.e., a transaction resolves a conflict by aborting itself.

VWIR Design. The first design is similar to DSTM [1] and TinySTM [8]. 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 $x.writer$. The limitations of this design are shown by giving a class of common inputs that it never accepts. For instance, the input pattern depicted in Figure 1 may arise when concurrent operations (searches, insertions) are executed on a linked list. All proofs are deferred to the technical report [9].

Theorem 1. *There is no TM implementing VWIR that accepts any input pattern of the following class:* $\mathcal{C}1 = \pi^*(r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \mid w_j^x \neg c_j^* r_i^x)\pi^*$.

IWIR Design. Here, we outline a second design that accepts patterns of the preceding class, i.e., for which the previous impossibility result does not hold. This design, inspired by WSTM [10] and TL2 [11], uses invisible writes and invisible reads with a lazy acquire technique that postpones effects until commit-time, thus it is called *IWIR*. Even IWIR design does not accept some very common input patterns. 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. 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 2. *There is no TM implementing IWIR that accepts any input pattern of the following class:* $\mathcal{C}2 = \pi^*(r_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* r_i^x) \neg c_i^* c_j \pi^*$.

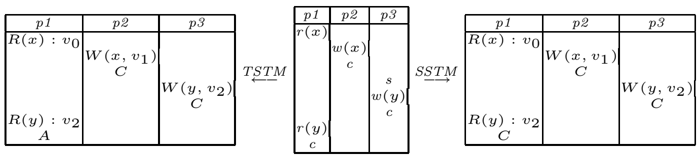

Fig. 2. An input pattern (in the center) that TSTM does not accept as described on the left-hand side. The commit-abort ratio obtained for TSTM is $\tau = \frac{2}{3}$ (transactions of *p2* and *p3* commit but transaction of *p1* aborts with Polite). In contrast, the Serializable STM presented in Section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

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. This design is inspired by the recently proposed TSTM [7] in its single-version mode.

TSTM is claimed to achieve conflict-serializability, however, it does not accept all possible conflict-serializations. Figure 2 (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 transaction of $p1$ 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. *There is no TM implementing CTR that accepts any input pattern of the following class:*

$$\mathcal{C}3 = (\neg w^x)^* r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \neg c_i^* s_k \neg (c_i \mid c_k \mid r_k^x)^* w_k^y \neg (c_i \mid c_k \mid r_k^x)^* c_k \neg c_i^* r_i^y \pi^*.$$

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

4 SSTM, an Implementation of the RTR Design

We propose a new design, the *Real-Time Relaxation (RTR)* design, 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.

SSTM, standing for *Serializable STM*, implements the RTR design and presents a high input acceptance. Moreover, SSTM is conflict-serializable but not opaque (SSTM accepts a history that is not opaque as illustrated on the right-hand side of Figure 2) and it avoids cascading abort, since whenever a transaction t_1 reads a value from another transaction t_2, t_2 has already committed [12]. Finally, SSTM is also fully decentralized, i.e., it does not use global parameters as opposed to other serializable STMs [6,7] that may experience congestion when scaling to large numbers of cores. Figure 1 presents the pseudocode of SSTM. For the sake of clarity of the presentation, we assume in the pseudocode of the algorithm that each function is atomic and we do not specify how shared variables are updated. We refer to T, X, V, as the sets of transaction identifiers, variable identifiers, and variable values, respectively.

Algorithm 1. SSTM – Serializable STM

```
**State of transaction** $t$:
    $status \in \{\text{active}, \text{inactive}\}$, initially active
    $read\text{-}set \subset X$, initially $\emptyset$
    $write\text{-}set \subset X \times V$, initially $\emptyset$
    $invisible\text{-}reads \subset X$, initially $\emptyset$
    $cr \subset T$, initially $\emptyset$ // set of concurrent readers

**State of variable** $x$:
    $read\text{-}fc \subset T$, initially $\emptyset$ // read future conflicts
    $write\text{-}fc \subset T$, initially $\emptyset$ // write future conflicts
    $active\text{-}readers \subset T$, initially $\emptyset$
    $val \in V$, initially the default value

**commit()**$_t$:
    **for** all $x \in read\text{-}set$ **do**
        $x.read\text{-}fc \leftarrow x.read\text{-}fc \cup \{t\}$
    **for** all $\langle x, t' \rangle$ such that $x \in read\text{-}set \wedge t' \in x.write\text{-}fc \vee \langle x, * \rangle \in write\text{-}set \wedge t' \in x.write\text{-}fc \cup x.read\text{-}fc$ **do**
        **for** all $r' \in t'.cr$ **do**
            **if** $r'.status =$ inactive **then**
                $t'.cr \leftarrow t'.cr \setminus \{r'\}$
                **if** $t'.cr = \emptyset$ **then**
                    $x.write\text{-}fc \leftarrow x.write\text{-}fc \setminus \{t'\}$
                    $x.read\text{-}fc \leftarrow x.read\text{-}fc \setminus \{t'\}$
            **else if** $r' = t$ **then** abort()
            **else** $cr \leftarrow cr \cup \{r'\}$
    $status \leftarrow$ inactive
    **for** all $\langle x, * \rangle \in write\text{-}set$ **do**
        **for** all $r \in x.active\text{-}readers$ **do**
            $cr \leftarrow cr \cup \{r\}$
            $x.read\text{-}fc \leftarrow x.read\text{-}fc \cup \{r\}$
        $x.write\text{-}fc \leftarrow x.write\text{-}fc \cup \{t\}$
    **for** all $\langle x, v \rangle \in write\text{-}set$ **do**
        $x.val \leftarrow v$
    clean()

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

**read**$(x)_t$:
    **if** $\langle x, v' \rangle \in write\text{-}set$ **then**
        $read\text{-}set \leftarrow read\text{-}set \cup \{x\}$
        $v \leftarrow v'$
    **else**
        $invisible\text{-}reads \leftarrow invisible\text{-}reads \cup \{x\}$
        $x.active\text{-}readers \leftarrow x.active\text{-}readers \cup \{t\}$
        **for** all $t'$ in $x.write\text{-}fc$ **do**
            **for** all $r' \in t'.cr \wedge r'.status =$ active **do**
                **if** $r' = t$ **then** abort()
                **else**
                    $cr \leftarrow cr \cup \{r'\}$
                    $x.read\text{-}fc \leftarrow x.read\text{-}fc \cup \{t\}$
        $v \leftarrow x.val$
    return $v$

**abort()**$_t$:
    $status \leftarrow$ inactive
    clean()

**clean()**$_t$:
    **for** all $y \in invisible\text{-}reads$ **do**
        $y.active\text{-}readers \leftarrow y.active\text{-}readers \setminus \{t\}$
    **for** all $x$ such that $\langle x, * \rangle \in write\text{-}set$ or $x \in read\text{-}set$ **do**
        **for** all $t' \in x.write\text{-}fc \cup x.read\text{-}fc$ **do**
            **for** all $r' \in t'.cr$ **do**
                **if** $r'.status =$ inactive **then**
                    $t'.cr \leftarrow t'.cr \setminus \{r'\}$
                    **if** $t'.cr = \emptyset$ **then**
                        $x.write\text{-}fc \leftarrow x.write\text{-}fc \setminus \{t'\}$
                        $x.read\text{-}fc \leftarrow x.read\text{-}fc \setminus \{t'\}$
```

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 concurrent 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 $t.cr$ (the concurrent readers of transaction t). Transaction t detects those transactions either because they are in *x.active-readers* (Line 27) or because they precede a transaction t' that is in *x.write-fc* (the future conflicts caused by a write access to object x) and they appear in $t'.cr$ (Line 23).Instead of keeping track of the following transactions, transaction t makes sure that any transaction detects it and all its preceding transactions $t.cr$ by recording itself in *x.write-fc* (Line 29)or *x.read-fc* (Lines 47 and 14).

Transaction t may abort for two reasons. First, if a read operation cannot return a value without violating consistency (Line 44).Second, if there exists a transaction that t precedes (Lines 16, 22) but that also precedes t (Line 15).Finally, the clean function is dedicated to garbage collect by emptying the records (Lines 54–63). A transaction t is removed from the *write-fc* and *read-fc* sets only when all its preceding transactions have completed, i.e., their $t.cr = \emptyset$ (Lines 19–21).For the correctness proof, please refer to the full version of this paper [9].

Theorem 4. *SSTM is conflict-serializable.*

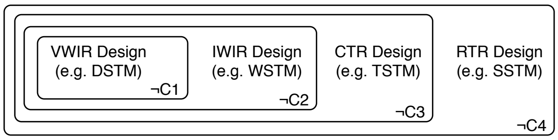

Fig. 3. Comparing the input acceptance of the TM designs. The VWIR design accepts no input patterns of the presented classes, the IWIR design accepts inputs that are neither in $\mathcal{C}1$ nor in $\mathcal{C}2$, and the CTR design accepts input patterns only outside $\mathcal{C}3$. Finally, we have not yet identified single-version patterns not accepted by design RTR.

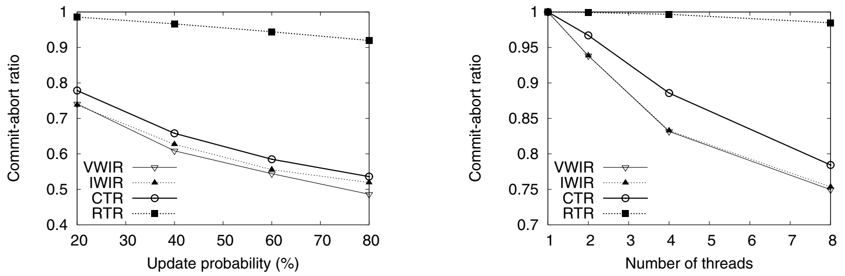

Fig. 4. Comparison of average commit-abort ratio of the various designs on a 256 element linked list: (left) with 8 threads as a function of the update probability; (right) with a 20% update probability as a function of the number of threads. As expected, the distinct commit-abort ratios follow the input acceptances but, interestingly, RTR presents a much better commit-abort ratio than other designs. Note that its heavy mechanism may produce an overhead compared to other designs.

5 Class Comparison and Experimental Validation

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. Let $\mathcal{C}4$ be the class of all possible input patterns. Given the input acceptance upper bound, we are able to draw the input acceptance of VWIR, IWIR, CTR, and RTR designs restricted to patterns that are in $\neg\mathcal{C}1$, $\neg\mathcal{C}2$, $\neg\mathcal{C}3$, and $\neg\mathcal{C}4$, respectively. The hierarchy shown in Figure 3 compares the input acceptance of these TM designs.

To validate experimentally the tightness of our bounds on the input acceptance of our TM designs, we have implemented and tested all these designs: VWIR, IWIR, CTR, and RTR on an 8-core Intel Xeon machine using a sorted linked list benchmark. Results are depicted on Figure 4.

6 Conclusion

We upper-bounded the input acceptance of well-known TM designs and we proposed a new TM design with a higher acceptance. 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. We expect this result to encourage further research on the best tradeoff between design simplicity and high input acceptance.

References

1. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003, pp. 92–101 (2003)
2. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
3. Guerraoui, R., Kapałka, M.: On the correctness of transactional memory. In: PPoPP 2008, pp. 175–184 (February 2008)
4. Yannakakis, M.: Serializability by locking. J. ACM 31(2), 227–244 (1984)
5. Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: PODC 2007, pp. 340–341 (2007)
6. Napper, J., Alvisi, L.: Lock-free serializable transactions. Technical Report TR-05-04, Department of Computer Sciences, University of Texas at Austin (2005)
7. Aydonat, U., Abdelrahman, T.S.: Serializability of transactions in software transactional memory. In: TRANSACT 2008. ACM, New York (2008)
8. Felber, P., Riegel, T., Fetzer, C.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008 (February 2008)
9. Gramoli, V., Harmanci, D., Felber, P.: Toward a theory of input acceptance for transactional memories. Technical Report LPD-REPORT-2008-009, EPFL (2008)
10. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not 38(11), 388–402 (2003)
11. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
12. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)