Theory Comput Syst
DOI 10.1007/s00224-015-9607-7

On Avoiding Spare Aborts in Transactional Memory

Idit Keidar · Dmitri Perelman

Abstract This paper takes a step toward developing a theory for understanding aborts in transactional memory systems (TMs). Existing TMs may abort many transactions that could, in fact, commit without violating correctness. We call such unnecessary aborts *spare aborts*. We classify what kinds of spare aborts can be eliminated, and which cannot. We further study what kinds of spare aborts can be avoided efficiently. Specifically, we show that some spare aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it reduces the number of spare aborts. We also present an efficient example TM algorithm that avoids certain kinds of spare aborts, and analyze its properties and performance.

Keywords Transactional memory

1 Introduction

The emergence of multi-core architectures raises the problem of efficient synchronization in multithreaded programs. Conventional locking solutions introduce a host of well-known problems: coarse-grained locks are not scalable, while fine-grained locks are error-prone and hard to design. Transactional memory [10, 16] has gained popularity in recent years as a new synchronization abstraction for multithreaded systems, which has the potential to overcome the pitfalls of traditional locking schemes. A *transactional memory* toolkit, or *TM* for short, allows threads to bundle

I. Keidar · D. Perelman (✉)
Department of Electrical Engineering, Technion, Haifa 32000, Israel
e-mail: dmitri.perelman@gmail.com

I. Keidar
e-mail: idish@ee.technion.ac.il

multiple operations on memory objects into one transaction. Similarly to database transactions [17], transactions are executed *atomically*: either all of the transaction's operations appear to take effect simultaneously (in this case, we say that the transaction *commits*), or none of transaction's operations are seen (in this case, we say that transaction *aborts*). We formally define the model and correctness criterion in Section 3.

A transaction's abort may be initiated by a programmer or may be the result of a TM decision. In the latter case, we say that the transaction is *forcefully aborted* by the TM. For example, when one transaction reads some object A and then writes to some object B, while another transaction reads the old value of B and then attempts to write A, one of the transactions must be aborted in order to ensure atomicity. Most existing TMs perform unnecessary (*spare*) aborts, i.e., aborts of transactions that could have committed without violating correctness; see Section 2. Spare aborts have several drawbacks: work done by the aborted transaction is lost, computer resources are wasted, and the overall throughput decreases. Moreover, after the aborted transactions restart, they may conflict again, leading to livelock and degrading performance even further.

The aim of this paper is to advance the theoretical understanding of TM aborts, by studying what kinds of spare aborts can or cannot be eliminated, and what kinds of spare aborts can or cannot be avoided efficiently. Specifically, we show that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it avoids spare aborts.

Previous work introduced two related notions: commit-abort ratio [6] and permissiveness [7]. The latter stipulates that if it is possible to proceed without aborts and still not violate correctness, no aborts should happen. However, while shedding insight on the inherent limitations of online TMs, these notions do not provide an interesting yardstick for comparing TMs. This is because under these measures, all online TMs inherently perform poorly for some worst-case workloads, as we show in Section 4.

In Section 5, we define measures of spare aborts that are appropriate for online TMs. Intuitively, our *strict online permissiveness* property allows a TM to abort some transaction only if not aborting any transaction would violate correctness. Unlike earlier notions, strict online permissiveness does not prevent the TM from taking an action that might lead to an abort in the future. Thus, the information available to the TM at every given moment suffices to implement strict online permissiveness. Clearly, this property depends on the correctness criterion the TM needs to satisfy. In this paper, we consider opacity [8] or slight variants thereof (see Section 3). In this context, strict online permissiveness prohibits aborting a transaction whenever the execution history is equivalent to some sequential one. We prove that strict online permissiveness cannot be satisfied efficiently by showing a reduction from the NP-hard *view serializability* [14] problem. We then define a more relaxed property, *online permissiveness*, which allows the TM to abort transactions if otherwise it would have to change the serialization order between already committed transactions. We show that online permissiveness also has inherent costs — it cannot be satisfied by a TM using *invisible* reads. Moreover, the information about a read should be exposed in shared memory immediately after the read operation returns.

In Section 6, we show a polynomial time TM protocol satisfying online permissiveness. The protocol maintains a precedence graph of transactions and keeps it acyclic. Unfortunately, we show that the graph must contain some committed transactions. But without removing any committed transactions, detecting cycles in the precedence graph would be impractical as it would induce a high runtime complexity. Hence, we define precise garbage collection rules for removing transactions from the graph. Even so, a naïve traversal of the graph would be costly; we further introduce optimization techniques that decrease the number of nodes traversed during the acyclity check.

Finally, we note that our goal is not to build a better TM, but rather to understand what can and what cannot be achieved, and at what cost. Future work may further explore the practical aspects of the complexity vs. spare-aborts tradeoffs; our conclusions appear in Section 7.

2 Related Work

Most existing TM implementations (e.g., [3–5, 9]) abort one transaction whenever two overlapping transactions access the same object and at least one access is a write. While easy to implement, this approach may lead to high abort rates, especially in situations with long-running transactions and contended shared objects. Aydonat and Abdelrahman [2] referred to this problem and proposed a solution based on a conflict serializability graph and multi-versioned objects in order to reduce the number of unnecessary aborts. However, their solution still induces spare aborts, and does not characterize exactly when such aborts are avoided. Moreover, they implement a stricter correctness criterion than opacity, which inherently requires more aborts. Riegel et al. [15] looked at the problem of spare aborts from a different angle, and introduced weaker correctness criteria, which allow TMs to reduce the number of aborts.

Napper and Alvisi [13] described a serializable TM, based upon multi-versioned objects, which used cycle detection in the precedence graph when validating the correctness criterion. The focus of the paper was providing a lock-free solution. The authors did not refer to the aspect of spare aborts and, in fact, their TM did lead to spare aborts due to a limitation on write operations, which must insert the new version after the latest one. In addition, Napper and Alvisi did not refer to the problems of garbage collection and computational complexity of operations.

Gramoli et al. [6] referred to the problem of spare aborts and introduced the notion of *commit-abort ratio*, which is the ratio between the number of committed transactions and the overall number of transactions in the run. Clearly, the commit-abort ratio depends on the choice of the transaction that should be aborted in case of a conflict. This decision is the prerogative of a contention manager [9]. Attiya et al. [1] showed a $\Omega(s)$ lower bound for the competitive ratio for transactions' makespan of any online deterministic contention manager, where s is the number of shared objects. Their proof, however, does not apply to our model, because it is based upon the assumption that whenever multiple transactions need exclusive access to the same shared object, only one of these transactions may continue, while others should be immediately aborted. In contrast, our model allows the TM to postpone the decision regarding which transaction should be aborted till the commit, thus introducing additional knowledge and improving the competitive ratio. In this paper, we show that no TM can obtain a commit-abort ratio achieved by an optimal offline algorithm. This result suggests that it is not interesting to compare (online) TMs by their commit-abort ratio, as the distance from the optimal result turns out to be an artifact of the workload rather than the algorithm, and every TM has a workload on which it performs poorly by this measure.

Input acceptance is also a notion presented by Gramoli et al. [6] — a TM *accepts* a certain input pattern (sequence of operational invocations) if it commits all of its transactions. The authors compared different TMs according to their input acceptance patterns. Guerraoui et al. [7] introduced the related notion of π*-permissiveness*. Informally, a TM satisfies π-permissiveness for a correctness criterion π, if every history that does not violate π is accepted by the TM. Thus, π*-permissiveness* can be seen as optimal input acceptance. However, Guerraoui et al. focused on a model with single-version objects, and their correctness criterion was based upon conflict serializability, which is stronger than opacity and thus allows more aborts. They ruled out the idea of ensuring permissiveness deterministically, and instead provide a randomized solution, which is always correct and avoids spare aborts with some positive probability. In contrast, we do not limit the model to include single-version objects only, our correctness criterion is a generalization of *opacity* [8], and we focus on deterministic guarantees. Although permissiveness does not try to regulate the decisions of the contention manager, we show that no online TM may achieve permissiveness. Intuitively, this results from the freedom of choice for returning the object value during the read operation — returning the wrong value might cause an abort in subsequent operations, which is avoided by a clairvoyant (offline) algorithm.

3 Preliminaries and System Model

Our model definitions are based on [11].

Transactions A transaction consists of a sequence of transactional operations, where each operation is comprised of an invocation step and a subsequent matching response step, collectively called transactional steps. The system contains a set of transactional objects. Each transactional operation either accesses a transactional object, or tries to commit or abort the transaction. More precisely, let T_i be a transaction, o be a transactional object, and v be a value. Then a transactional operation is one of the following: (1) An invocation step *start*(T_i), followed by response S, meaning T is started. (2) An invocation step *read*(T_i, o), followed by a response step that either gives the current value of o, or responds A, meaning that the transaction is aborted. (3) An invocation *write*(T_i, o, v), followed by a response either acknowledging the write, or responding A. For simplicity, we assume that all the written values are unique. (4) An invocation *tryAbort*(T_i), followed by response A

(abort operation). (5) An invocation *tryCommit*(T_i), followed either by response C, meaning T committed, or A. We say the *read set* (resp. *write set*) of a transaction is the set of transactional objects read by (resp. written to) T, they are not known in advance.

Transaction histories A *transactional history* H is a sequence of transactional steps, interleaved in an arbitrary order (in the rest of the paper we use the notion of *run* as a synonym to a transactional history). A transaction is *active* in H if it is neither committed nor aborted, it is *complete* otherwise. A transaction can perform operations as long as it is active. Each transaction has a unique identifier (id). Retrying an aborted transaction is interpreted as creating a new transaction with a new id.

Two histories H_1 and H_2 are *equivalent* if they contain the same transactions and each transaction T_i issues the same operations in the same order with the same responses in both. A history H is *complete* if it does not contain active transactions. If history H is not complete, we may build from it a complete history *Complete(H)* by adding an abort operation for every active transaction. We define *committed(H)* to be the subsequence of H consisting of all the operations of all the committed transactions in H.

The real-time order on transactions is as follows: if the first operation of transaction T_i is issued after the last response of a complete transaction T_j in H, then T_j precedes T_i in H, denoted $T_j \prec_H T_i$. Transactions T_i and T_j are concurrent if neither $T_j \prec_H T_i$, nor $T_i \prec_H T_j$. A history S is sequential if it has no concurrent transactions. A sequential history S is legal if it respects the sequential specification of each object accessed in S. Transaction T_i is legal in S if the largest subsequence S' of S, such that for every transaction $T_k \in S'$, either (1) $k = i$, or (2) T_k is committed and $T_k \prec_S T_i$, is a legal history.

Transactional memory A *transactional memory (TM)* is an algorithm for running transactions. We do not consider any kind of transactional nesting. In this paper, we assume the algorithm consists of a set of *threads*. The threads communicate with each other using *shared memory*. Each transaction is *run* by a thread, and each thread runs at most one transaction at a time. To run a transaction T, a thread runs each of T's transactional operations, as follows. (1) Take as input an invocation step of T. (2) Perform a sequence of shared memory steps, which are determined by the input and the memory. (3) Return as output a response step to T. A transactional memory can forcefully abort transaction T_j as a result of invocation step of another transaction T_i. In this case we say that T_j is aborted and the next operation invocation of T_j returns A.

We call the memory objects accessed by the threads *base objects*. Note that these are conceptually distinct from the transactional objects accessed by the transactions. We also call the steps performed by the threads *base steps*. We assume that all the base steps for running a transactional step appear to execute atomically. The means by which such linearizability of transactional steps is achieved lies beyond the focus of our paper. In practice, it can be achieved using locks (like the two-phase locking mechanism used in commit operations by TL2 [3]), or by lock-free algorithms [5].

Due to the assumption of atomicity of transactional steps we consider only the well-formed histories in which an invocation of transactional operation is followed by the corresponding response.

We say that an STM is *responsive* if it guarantees that each operation invocation eventually gets a response, even if all other threads do not invoke new transactional operations. This limits the responsive STM's behavior upon operation invocation, so that it may either return an operation response, or abort a transaction, but cannot wait for other transactions to invoke new transactional operations. Note that we do allow for a responsive STM to wait for concurrent transactional operations to complete, for example TL2 [3] is responsive in spite of its use of locks. One may say that a responsive STM provides lock-freedom at the level of transactional operations. In this paper all the results are applied for responsive STMs only.

Correctness Our correctness criterion resembles the *opacity* condition of Guerraoui and Kapalka [8]. For a history H, and a partial order P on the transactions that appear in H, we say that H satisfies P-opacity if there exists a sequential history S such that:

- S is equivalent to *Complete(H)*.
- Every transaction $T_i \in S$ is legal in S.
- If $(T_i, T_j) \in P$, then $T_i \prec_S T_j$.

Given a function Γ that maps histories to partial orders of transactions that appear in those histories, we say a TM satisfies Γ-opacity if every history H generated by the TM satisfies $\Gamma(H)$-opacity.

When $\Gamma(H)$ is the real-time order on all the ordered pairs of non-concurrent transactions in H, the history S should preserve the real-time order of H as in the original definition of opacity. On the other hand, when $\Gamma(H)$ is empty, the correctness criterion is a serializability with consideration of aborted transactions. The use of Γ makes it possible to require a transactional ordering that lies between serializability and strict serializability according to any arbitrary rule (e.g., Riegel et al. [15] considered demanding real-time order only from transactions belonging to the same thread). We define a more general criterion in order to broaden the scope of our results. In the rest of this paper, we will assume that $\Gamma(H)$ is a subset of the real-time order on transactions, unless stated otherwise.

4 Limitations of Previous Measures

4.1 Commit-Abort Ratio

The *commit-abort ratio*, τ, [6] is the ratio between the number of committed transactions and the overall number of transactions in the history. Unfortunately, we now show that no online TM may guarantee an optimal commit-abort ratio.

We use the style of [15] to depict transactional runs. Objects are represented as horizontal lines o_1, o_2, etc. Transactions are drawn as polylines with circles corresponding to accesses to the objects. Filled circles indicate writes, and empty circles indicate reads. Commit is indicated by the letter **C**, and abort by the letter **A**. If the

TM implements the access to the object as if it had appeared in past, the dashed arrow indicates the point in time at which the access to the object appears according to the TM serialization.

As we said earlier, a TM does not know read and write accesses in advance, i.e., a TM is *online*. As opposed to this, we say that an *offline* algorithm knows the sequence of accesses of the transaction beforehand.

Lemma 1 *No TM can achieve the commit-abort ratio of an optimal offline algorithm.*

Proof Consider the scenarios depicted in Fig. 1. We show that no TM can achieve commit-abort ratio better than $\frac{1}{3}$ in both runs, while an optimal offline algorithm achieves $\tau = \frac{2}{3}$. Transactions T_1 and T_2 cannot both commit because they both write to o_1 after reading its previous value. There are three possible scenarios for a TM algorithm: 1) abort T_1, 2) abort T_2, or 3) abort both T_1 and T_2. Clearly, in the third case τ cannot be better than $\frac{1}{3}$.

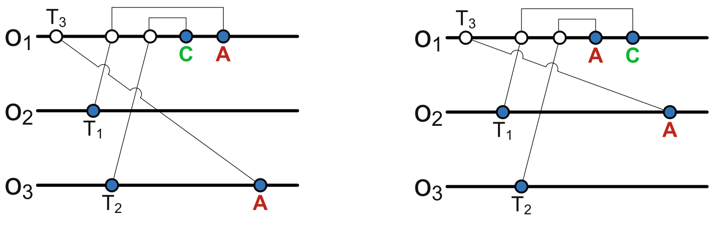
In run r_1 (Fig. 1a), the TM commits T_2 and T_1 is aborted. Then the adversary causes transaction T_3 to read o_3 — it must be aborted because it conflicts with T_2, resulting in $\tau = \frac{1}{3}$.

In run r_2 (Fig. 1b), the TM commits T_1 and T_2 is aborted. In this case the adversary causes T_3 to read o_2, T_3 must be aborted because of its conflict with T_1, resulting again in $\tau = \frac{1}{3}$.

Note that the optimal offline TM in these cases would abort only one transaction, yielding $\tau = \frac{2}{3}$. □

4.2 Permissiveness

Since requiring an optimal commit-abort ratio is too restrictive, we consider a weaker notion that limits aborts only in runs where none are necessary. Recall that a TM accepts a certain history if it commits all of its transactions. A TM provides *π-permissiveness* [7] if it accepts every history satisfying π (a TM provides Γ-opacity-permissiveness if it accepts every history satisfying Γ-opacity). Gramoli et



(a) Run r_1: T_2 commits, all other transactions abort: $\tau = \frac{1}{3}$

(b) Run r_2: T_1 commits, all other transactions abort: $\tau = \frac{1}{3}$

Fig. 1 No online TM may know whether to abort T_1 or T_2 in order to obtain an optimal commit-abort ratio

al. showed that existing TM implementations do not accept all inputs they could have, and hence are not π-permissive. We show that this is an inherent limitation.

The formal impossibility illustrated in Fig. 2 is captured in the following lemma:

Lemma 2 *For any Γ, there is no online TM implementation providing Γ-opacity-permissiveness.*

Proof Consider the scenario depicted in Fig. 2. All the objects have initial values, v_0. All the transactions start at the same time, and are therefore not ordered according to the real-time order, thus the third condition of our correctness criterion holds for any Γ because we assume that $\Gamma(H)$ is a subset of the real-time order of H.

T_1 writes values v_1 to o_1 and v_2 to o_2. At time t_0, there is a read operation of T_2 and the TM should decide what value should be returned. In general, the TM has four possibilities: (1) return v_1, (2) return v_0, (3) return some value v' different from v_0 and v_1, and (4) abort T_2. If the TM chooses to abort, then Γ-opacity-permissiveness is violated and we are done. (3) is not possible, for returning such a value would produce a history, for which any equivalent sequential history S would violate the sequential specification of o_1 and thus would not be legal.

Consider case (1): the TM returns v_1 for T_2 at time t_0. This serializes T_2 after T_1. Consider run r_1 depicted in Fig. 2a, where T_3 tries to write to o_3 and commit. In this run, the TM has to forcefully abort T_3, because not doing so would produce a history H with no equivalent sequential history: $T_1 \prec T_2 \prec T_3 \prec T_1$. However, if T_2 would read v_0 in run r_1, then T_2, T_1 and T_3 would be legal, and no transaction would have to be forcefully aborted. So Γ-opacity-permissiveness is violated.

In case (2), the TM returns v_0 for transaction T_2 at time t_0, serializing T_2 before T_1. Consider run r_2 depicted in Fig. 2b. Transaction T_4 writes to o_2, and afterwards reads and writes to o_3. Transaction T_4 has to be serialized after T_1, because T_1 has read v_0 from o_2. When T_2 tries to read and write to o_3 and commit, T_2 has to be serialized after T_4 because they both read and write to o_3. Therefore, the TM will have to forcefully abort some transaction, because not doing so would produce a history with no equivalent sequential history: $T_2 \prec T_1 \prec T_4 \prec T_2$. But if T_2 would read v_1 in run r_2, then no transaction would have to be forcefully aborted. So again, Γ-opacity-permissiveness is violated.

Runs r_1 and r_2 are indistinguishable to the TM at time t_0. Therefore, no online TM can accept both of the patterns, while an offline optimal TM can accept both of them. □

5 Online Permissiveness: Limitations and Costs

5.1 Strict Online Opacity-Permissiveness

Definition 1 Consider a history H, in which a transaction T receives an abort response A to one of its operations op. We say that H' is a live-T modification of H if H' is the same as H except that T receives a non-abort response to op in H'.

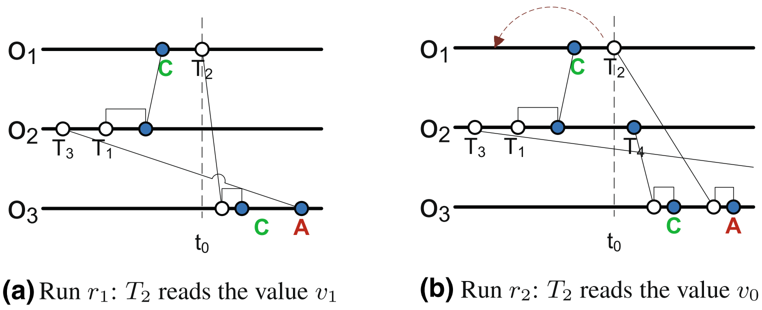
We now define a property that prohibits unnecessary aborts, and yet is possible to implement.

Definition 2 A responsive TM satisfies strict online Γ-opacity-permissiveness for a given Γ if the TM forcefully aborts a transaction T in a history H only if there exists no live-T modification of H that satisfies $\Gamma(H)$-opacity.

Note that this property does not define which transaction should be aborted if abort happens, and does not prohibit returning a value that will cause aborts in the future. For example, in the scenarios depicted in Fig. 2, at time t_0, a TM satisfying this property may return either value, even though this might cause an abort in the future.

An algorithm satisfying strict online opacity-permissiveness should be able to detect whether returning a given value creates a history satisfying Γ-opacity. We show that this cannot be detected efficiently. To this end, we recall a well-known result about checking the serializability of the given history, which was proven by Papadimitriou [14].

Given history H, the *augmented* history $\bar{H}$ is the history that is identical to H, except two additional transactions: T_{init} that initializes all variables without reading any, and T_{read} that is the last transaction of $\bar{H}$, reading all variables without changing them. The set of *live* transactions in H is defined recursively in the following way: (1) T_{read} is live in H, (2) If for some live transaction T_j, T_j reads a variable from T_i (every written value is unique according to our model), then T_i is also live in H. Note that aborted transaction cannot be live since no transaction may read a value written by an aborted one. A transaction is *dead* if it is not live. Two histories H and H' are *view equivalent* if and only if (1) they have the same sets of live transactions and (2) T_i reads from T_j in H if and only if T_i reads from T_j in H'. Note that a definition of view equivalence differs from a history equivalence defined in this paper, which demands the same order of operations for each transaction in equivalent histories. History H is *view serializable*, if for every prefix H' of H, *complete*(H') is view equivalent to some serial history S. The following is proven in [14]:



(a) Run r_1: T_2 reads the value v_1 **(b)** Run r_2: T_2 reads the value v_0

Fig. 2 At time t_0, no online TM knows which value should be returned to T_2 when reading o_1 in order to allow for commit in the future

Theorem 1 (Papadimitriou) *Testing whether the history H is view-serializable is NP-complete in the size of the history, even if H has no dead transactions.*

Lemma 3 *For any Γ, detecting whether the history H satisfies Γ-opacity is NP-complete in the size of the history.*

Proof We first note that the problem of detecting view serializability has a trivial reduction to the problem of identifying whether a given history H is view equivalent to some serial history S. Hence, in order to prove the claim, we need to show a reduction from the problem of detecting whether a history H is view-equivalent to some serial history S to the problem of detecting whether some history H' satisfies Γ-opacity. Consider history H with no dead transactions. Given the assumption of unique write values and in the absence of aborted transactions, the definition of view equivalence differs from the definition of opacity only in the fact that opacity refers to the partial order Γ, which is a subset of a real-time order. We construct history H', which is identical to history H except the following addition: for each T_i in H, we add $start(T_i)$ at the beginning of H'. We will show that H is view equivalent to some serial history S if and only if H' satisfies Γ-opacity.

All the transactions in H' are concurrent ($start(T_i)$ happens before any other operation of T_j for every T_i and T_j), therefore the third condition of Γ-opacity vacuously holds for any Γ. In the absence of aborts in H', H' satisfies Γ-opacity if and only if there exists a legal sequential history S', so that every transaction in H' issues the same invocation events and receives the same response events as in S'. Therefore, H' satisfies Γ-opacity if and only if H' is view-equivalent to some serial history S'. □

5.2 Online Opacity-Permissiveness

Intuitively, the problem with strict online opacity-permissiveness lies in the fact that the order of committed transactions may be undefined and may change in the future. Consider, for example, the scenario depicted in Fig. 3. Transactions T_1 and T_2 are not ordered according to real-time order, therefore they are not ordered by Γ. At time t_0, the serialization order is $T_1 \rightarrow T_2$, as o_1 holds the value written by T_2. When T_3 commits, the serialization order of T_1 and T_2 becomes undefined, since

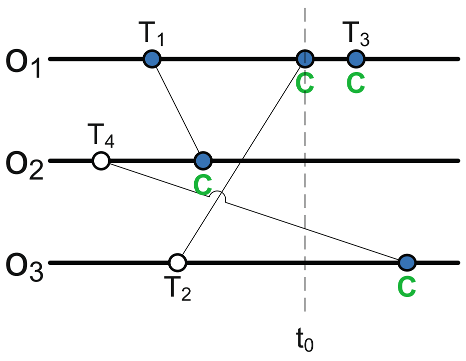


Fig. 3 The order of transactions T_1 and T_2 is changed after their commit time

T_3 overwrites o_1 before any transaction reads the value written by T_2. And when T_4 commits, the serialization order becomes $T_2 \to T_4 \to T_1 \to T_3$. If the partial serialization order induced by the run cannot change after being defined, the problem becomes much easier. To capture this restriction, we extend the TM's interface so as to make the serialization decisions explicit: every commit operation returns a partial order on all committed transactions with conflicting writes. Specifically, we assume that a successful tryCommit(T_i) operation returns, instead of C, a partial order R_i on previously committed transactions.

We denote by $R(t)$ the value returned in the last commit occurring by point t in H; $R(t)$ is empty if no commit occurs by time t in H.

Note that this interface is only intended to expose the internal state of the TM, in order to facilitate reasoning, and can be filtered out before actually providing a response to the application. Using this interface, we now define the persistent ordering property, which prevents a TM from "changing its mind" about the serialization order of already committed conflicting transactions.

Definition 3 (Persistent Ordering) A history H (with the modified interface) satisfies persistent ordering if it satisfies all of the following: 1) $R(t)$ orders all pairs of transactions T_i and T_j that have committed by point t with intersecting write-sets. 2) For all t' and t such that $t' < t$, $R(t') \subseteq R(t)$. 3) H satisfies $R(t)$-opacity for all t.

In other words, if committed transactions T_i and T_j both write to the same object in H, then they are explicitly ordered by the time both of them commit and their order persists thereafter. We say that a TM satisfies Persistent Ordering if every history generated by the TM satisfies Persistent Ordering. We now define our more relaxed property, online Γ-opacity-permissiveness, which may be satisfied at a polynomial cost.

Definition 4 A responsive TM satisfies online Γ-opacity-permissiveness for a given Γ if:

1. The TM satisfies Persistent Ordering.
2. The TM forcefully aborts a transaction T in a history H only if there exists no live-T modification of H that satisfies persistent ordering and $\Gamma(H)$-opacity.

Note that Definition 4 implies that each committing transactions should define its serialization order with regard to all other committed transactions that have written to the same objects. To the best of our knowing, all existing TMs do in fact define the order on two transactions that write to the object by the time the latter transaction commits. We note that this requirement might be limiting for TMs that wish to exploit the benefits of commutative or write-only operations (see [12]), and do not necessarily define the serialization point of the committed transactions. However, this limitation is essential for an effective check of the opacity criterion.

In the following sections we show a polynomial-time TM satisfying online opacity-permissiveness. We now prove that such an implementation, nevertheless, has some inherent costs.

Impossibility of invisible reads One of the basic decisions that needs to be made during the design of a TM is whether to *expose* the fact that transaction T_i has read the object o, i.e., make a change in shared memory as a result of the read, making the read *visible*. In case we expose the read, there arises another question, regarding whether we can postpone exposing the read until the commit. One of the central problems with exposing the read is that it requires writing metadata in shared memory. One typically tries to avoid writes to shared memory, because writing data that is read by different cores has a high cache penalty. Postponing exposing the read until the commit may save redundant writes in case the transaction eventually aborts.

Unfortunately, we shall now show that if the serialization order can violate the real-time order of transactions, then online opacity permissiveness requires all reads to be exposed in shared memory immediately after a read happens. To this end, we first need to rule out trivial TMs, for example, ones that always return an object's initial value in response to a read. We formally define our non-triviality requirement as follows:

Definition 5 For a given Γ, a TM is non-trivial if a read operation of object o by transaction T_i does not return an older value than the last one written to o by a committed transaction before T_i began, unless returning the last value written before T_i began generates a history H that is not $\Gamma(H)$-opaque.

In other words, read may return an old value only if there is a good reason to do so (avoiding an abort). We now show that every non-trivial TM satisfying online opacity-permissiveness with no respect to real-time order must expose all its read operations immediately as they happen:

Lemma 4 *Let $\Gamma_\emptyset$ be a function from histories to partial transactional orders such that $\Gamma_\emptyset(H) = \emptyset$. If a non-trivial responsive TM satisfies online $\Gamma_\emptyset$-opacity-permissiveness, then any active transaction T_i that has read $n \geq 2$ distinct objects must keep all its reads visible.*

Proof Assume by contradiction that there exists a non-trivial TM satisfying online Γ-opacity-permissiveness and there exists an active transaction T_r that reads non-initial values of objects o_1 and o_2 (and perhaps some additional objects) until time t_0 and does not expose the read of some object o_1, as depicted in the left part of Fig. 4a.

We now continue the run from t_0 onward as described below. We invoke transaction T_1 reading object o_3, and then transaction T_2 that reads o_3, writes to o_3 and reads o_2. By non-triviality, T_1 and T_2 read the same version of o_3, hence once T_2 writes to o_3, T_2 is serialized after T_1. Moreover, T_2 must read the version of o_2 written by T_{w2} — the same one as read by T_r. We next invoke transaction T_3, which reads o_4. We then continue transaction T_1 so that it writes to o_2, then reads o_4, writes to o_4 and commits. As mentioned earlier, T_1 is serialized before T_2 and T_2 reads the object version written by T_{w2}, therefore T_1 must be serialized before T_{w2} (and before T_r). Note that T_1 can be serialized before T_{w2} because $\Gamma_\emptyset$ does not impose a real-time order on transactions. By non-triviality, T_1 and T_3 read the same version of o_4, hence T_3 is serialized before T_1 (and before T_r).

Finally, we continue transaction T_3 so that it reads o_1, writes to o_1 and tries to commit. By non-triviality, T_3 reads the version of o_1 written by T_{w1}. The commit operation of T_3 cannot succeed: on the one hand, T_3 must be serialized after T_{w1}, and on the other hand T_3 must be serialized before T_r, but T_3 cannot be serialized between T_{w1} and T_r because T_r reads the version of o_1 written by T_{w1}. Hence T_3 aborts in r_1.

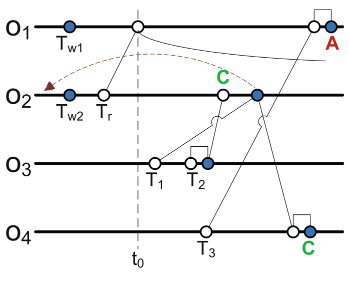
Consider run r_2 depicted in Fig. 4b. This run is identical to r_1 except that T_r's read of o_1 is removed. T_3 can commit successfully in r_2, with the following serialization order: $\{T_{w1}, T_3, T_1, T_{w2}, T_r, T_2\}$. Since we assume the TM satisfies online Γ-opacity-permissiveness, T_3 commits. However, since T_r does not expose its read of o_1, T_3 cannot distinguish between r_1 and r_2, a contradiction. □

6 The AbortsAvoider Algorithm

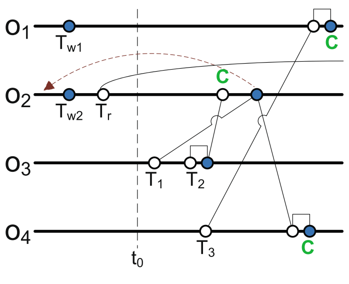
We now present AbortsAvoider, a TM algorithm implementing online opacity-permissiveness for any given Γ. The basic idea behind AbortsAvoider is to maintain a precedence graph of transactions, and keep it acyclic, as explained in Section 6.1. A simplified version of the protocol based on this graph is then presented in Section 6.2. A key challenge AbortsAvoider faces is that completed transactions cannot always be removed from the graph, whereas keeping all transactions forever is clearly impractical. We address this challenge in Section 6.3, presenting a garbage collection mechanism for removing terminated transactions from the graph. In Section 6.4 we present another optimization, which shortens paths in the graph to reduce the number of terminated transactions traversed during the acyclity check. Our complexity analysis appears in the same section.

6.1 Basic Concept: Precedence Graph

Information bookkeeping Our protocol maintains *object version lists*. We now explain what such a TM does: (1) each object o is associated with a totally ordered



(a) Run r_1: T_3 cannot commit



(b) Run r_2: T_3 commits

Fig. 4 T_3 does not distinguish between r_1 and r_2 at time t_0 if T_r does not expose its read set of versions, (2) a read of o returns the value of one of o's versions, and (3) a write to o adds a new version of o upon commit. For simplicity, at any given moment, we number the versions of the object in increasing order. (Note that the numbering is for analysis purposes only, and the numbers of the versions change during the run as the versions are inserted and removed from the versions list). The object version $o.v_n$ includes the data, $o.v_n.data$, the writer transaction, $o.v_n.writer$, and a set of readers, $o.v_n.readers$. Each transaction has a *readList* and a *writeList*. An entry in a *readList* points to the version that has been read by the transaction. A *writeList* entry points to the object that should be updated after commit, the new data, and the place to insert the new version, (which may be undefined till the commit). For the sake of simplicity we assume that the values written to transactional objects are unique.

Precedence graph Transactions may point to one another, forming a directed labelled precedence graph, PG. PG reflects the dependencies among transactions as created during the run. We denote a precedence graph of history H as PG_H. The vertices of PG are transactions, the edges of PG are as follows (Fig. 5):

If $(T_j, T_i) \in \Gamma(H)$, then PG contains (T_j, T_i) labelled L_Γ (Γ order). If T_i reads $o.v_n$ and T_j writes $o.v_n$, then PG contains (T_j, T_i) labelled L_{RaW} (Read after Write). If transaction T_i writes $o.v_n$ and T_j writes $o.v_{n-1}$, then PG contains (T_j, T_i) (Write after Write) labelled L_{WaW}. If transaction T_i writes $o.v_n$ and T_j reads $o.v_{n-1}$, then PG contains (T_j, T_i) labelled L_{WaR} (Write after Read).

Below we present lemmas that link maintaining acyclity in PG and satisfying online-permissiveness. To this end, we restrict our discussion to non-local histories, which we now define. We say that a read operation of T_i $read_i(o)$ in H is *local* if it is preceded in $H|T_i$ by a write operation $write_i(o, v)$. A write operation $write_i(o, v)$ is *local* if it is followed in $H|T_i$ by another write operation $write_i(o,v')$. The *non-local* history of H is the longest subsequence of H not containing local operations [8]. Note that the precedence graph does not refer to local operations.

We denote $PG(t)$ to be the graph at time t. We define λ_{PG} to be the following binary relation: if PG contains a path from T_i to T_j consisting of L_{WaW} edges, then $T_i \prec_{\lambda_{PG}} T_j$. Note that if PG is acyclic, then λ_{PG} is reflexive, antisymmetric and transitive, and therefore λ_{PG} is a partial order.

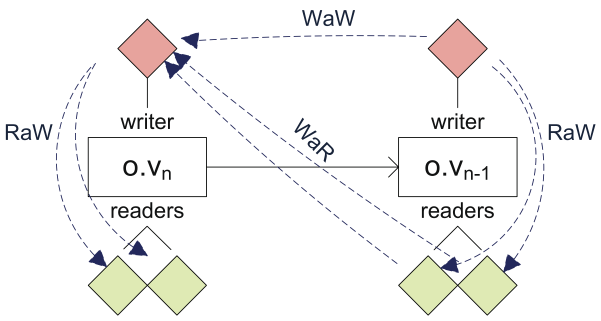


Fig. 5 Object versions and the precedence graph, PG

Lemma 5 *Consider a TM maintaining object version lists. If PG is acyclic throughout some run, then the non-local history H of the run satisfies $\Gamma \cup \lambda_{PG}$-opacity.*

Proof Let H be a history over transactions $\{T_1 \dots T_n\}$. Let $H_C = Complete(H)$, i.e. H with A_i added for every active $T_i \in H$.

Since PG is acyclic, it can be topologically sorted. Let $T_{i1}, \dots, T_{in}$ be a topological sort of PG, and let S be the sequential history $T_{i1}, \dots, T_{in}$. Clearly, S is equivalent to H_C because both of the histories contain the same transactions and each transaction issues the same operations and receives the same responses in both of them.

We now prove that every $T_i \in S$ is legal. Assume by contradiction that there are non-legal transactions in S. Let T_i be the first such transaction. If T_i is non-legal, T_i reads a value of object o that is not the latest value written to o in S by a committed transaction. (Recall that by definition of object version lists, only values written by committed transactions can be read). S contains only non-local operations, and therefore T_i reads the version $o.v_n$ written by another transaction T_j. Therefore, there is an edge from T_j to T_i in PG. It follows that T_j is committed in S and ordered before T_i according to the topological sort. If the value of $o.v_n$ is not the latest value written in S before T_i, then there exists another committed transaction T'_j that writes to o and is ordered between T_j and T_i in S. If T'_j writes to a version earlier than $o.v_n$, then there is a path from T'_j to T_j in PG, and therefore T'_j is ordered before T_j in S. If T'_j writes to a version later than $o.v_n$, then there is a path from T_i to T'_j in PG, and therefore T'_j is ordered after T_i in S. In any case, T'_j cannot be ordered between T_j and T_i in S, a contradiction.

For each pair $T_i \prec_\Gamma T_j$, PG contains an edge from T_i to T_j. Therefore, according to the topological sort, S preserves the partial order Γ. By definition S also preserves the order defined by λ_{PG}.

Summing up, *Complete*(H) is equivalent to a legal sequential history S, and S preserves partial order $\Gamma \cup \lambda_{PG}$. Therefore H is $\Gamma \cup \lambda_{PG}$-opaque. □

Lemma 6 *Every TM that maintains object version lists and keeps PG acyclic satisfies persistent ordering.*

Proof In order to prove that a TM satisfies persistent ordering we need to show the following: 1) define a partial order R_i returned by a successfully committed transaction (in other words, define the way a TM exports an ordering interface); 2) show that R_i orders all pairs of committed transactions with a non-empty intersection of their write-sets; 3) show that $R(t)$ monotonically increases with t and 4) prove that $H|_t$ satisfies $R(t)$-opacity at any t.

1) We define R_i returned by a successfully committed transaction at time t to be $\lambda_{PG(t)}$; in other words R_i orders T_i and T_j if they are connected in PG by L_{WaW} edges.
2) Consider two committed transactions T_k and T_m that have a common object o in their write-sets such that T_k has written to the version $o.v_i$ and T_m has written to

the version $o.v_j$, where$i < j$. In this case PG contains a path from T_k to T_m consisting of L_{WaW} edges and therefore λ_{PG} contains a pair (T_k, T_m). Hence, R_i orders all pairs of committed transactions with a non-empty intersection of their write-sets.

3) According to the rules for updating PG, L_{WaW} edges are never removed and $R(t') \subseteq R(t)$ for every $t' < t$.
4) According to Lemma 5, $H|_t$ satisfies $\lambda_{PG(t)}$-opacity and therefore $H|_t$ satisfies $R(t)$-opacity.

□

Lemma 7 *Consider a responsive TM maintaining object version lists and keeping PG acyclic. Consider that this TM forcefully aborts a transaction T in a history H only if there exists no live-T H's modification H', such that PG'_H contains no cycles. Then this TM satisfies online Γ-opacity-permissiveness.*

Proof As shown in Lemma 6, the TM satisfies persistent ordering. We need to show that if there is a cycle in PG, then the run violates $(\Gamma \cup \lambda_{PG})$-opacity.

We show first that if there is an edge (T_i, T_j) in PG, then every legal sequential history S preserving $\Gamma \cup \lambda_{PG}$ and equivalent to *Complete(H)* orders T_i before T_j. Consider two transactions T_i and T_j such that there is an edge (T_i, T_j) in PG. If the edge is labeled L_Γ, then $(T_i, T_j) \in \Gamma$, and S orders T_i before T_j. If the edge is labeled L_{RaW}, then T_j reads a value written by T_i and S also orders T_i before T_j. If the edge is labeled L_{WaW}, then $T_i < T_j$ according to λ_{PG}, hence S also orders T_i before T_j. If the edge is labeled L_{WaR}, then T_i reads $o.v_n$ while T_j writes $o.v_{n+1}$. On the one hand, T_j should be ordered after $o.v_n.writer$ in S (there is an edge from $o.v_n.writer$ to T_j labeled L_{WaW}). On the other hand, T_j cannot be ordered between $o.v_n.writer$ and T_i, because T_i must read the value written by $o.v_n.writer$ in S. Therefore, T_j is ordered after T_i in S in this case as well.

Summing up, an edge (T_i, T_j) in the precedence graph induces the order of T_i before T_j in any legal sequential history S preserving $\Gamma \cup \lambda_{PG}$ and equivalent to *Complete(H)*. Therefore, if PG contains a cycle, no such sequential history exists, and the TM cannot satisfy $\Gamma \cup \lambda$-opacity. □

Corollary 1 *Consider a TM maintaining object version lists that keeps PG acyclic. Consider that this TM forcefully aborts a transaction T in a history H only if there exists no live-T H's modification H', such that PG'_H contains no cycles. Then this TM satisfies Γ-opacity and online Γ-opacity-permissiveness.*

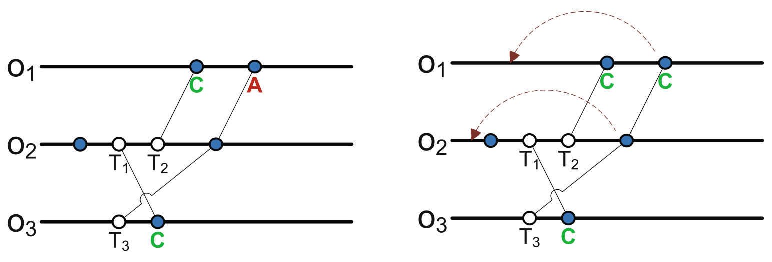
6.2 Simplified Γ-AbortsAvoider Algorithm

AbortsAvoider algorithm maintains object version lists as explained above, keeps PG acyclic and forcefully aborts a transaction only if not aborting any transaction would create a cycle in PG. Read and write operations are straightforward, they are depicted in Algorithm 1. A *read* operation (line 4) looks for the latest possible object version to read without creating a cycle in PG. *Write* operations (line 13) postpone the actual work till the commit.

Algorithm 1 Γ-AbortsAvoider for T_i - Read/Write.

```
1: procedure START()
2:     $prev = \{T_j : (T_j, T_i) \in \Gamma(H) \wedge \nexists T_k \in PG : (T_j, T_k) \in \Gamma(H)\}$
3:     $\forall T_{prev} \in prev : PG.\text{ADDEDGES}(\{(T_{prev}, T_i)\})$
4: procedure READ(Object o)
5:     if $o \in T_i.writeList$ then return $T_i.writeList[o].data$
6:     if $o \in T_i.readList$ then return $T_i.readList[o].data$
7:     $n \leftarrow$ the latest version that can be read without creating a cycle in $PG$
8:     if $n = \perp$ then return abort event $A_i$
9:     $PG.\text{ADDEDGES}(\{(o.v_n.writer, T_i), (T_i, o.v_{n+1}.writer)\})$
10:    $o.v_n.readers.\text{ADD}(T_i)$
11:    $T_i$.readList.ADD($\langle o.v_n \rangle$)
12:    return $o.v_n.data$
13: procedure WRITE(Object o, ObjectData val)
14:    if $o \in T_i.writeList$ then
15:        $T_i.writeList[o].data \leftarrow val$; return
16:    if $o \in T_i.readList$ then
17:        ▷ non-blind write, victim version is read version
18:        $writeNode \leftarrow \langle o, readList[o].version, val \rangle$
19:    else
20:        ▷ blind write, victim version is not known
21:        $writeNode \leftarrow \langle o, \perp, val \rangle$
22:    $T_i$.writeList.ADD($writeNode$)
```

The *commit* operation is more complicated. Intuitively, for each object written during transaction, the algorithm should find a place in the object's version list to insert the new version without creating a cycle. Unfortunately, checking the objects one after another in a greedy way can lead to spare aborts, as we illustrate in Fig. 6a. Committing T_3 first seeks for a place to install the new version of o_1 and decides to install it after the last one (serializing T_3 after T_2). When T_3 considers o_2, it discovers that the new version cannot be installed after the last one, because T_3 should precede T_1, but it also cannot be installed before the last one, because that would make T_3 precede T_2, so T_3 is aborted. However, installing the new version of o_1 before the last



(a) Run with greedy check **(b)** Run with no spare aborts

Fig. 6 Checking the written objects in a greedy way during the commit may lead to a spare abort

one would have allowed T_3 to commit, as depicted in Fig. 6b, that is why aborting T_3 violates online Γ-opacity-permissiveness.

Algorithm 2 Γ-AbortsAvoider for $\mathrm{T_i}$ - Commit.

```
23: procedure COMMIT
24:     newEdges ← $\emptyset$                                        ▷ edges added upon commit
25:     blinds ← $\emptyset$                                          ▷ the set of blind writes
26:     ▷ Phase I — install the non-blind writes
27:     for each n in $T_i$.writeList do
28:         if n.victim $\neq \perp$ then
29:             (v,edgs)←VALIDATEWRITE(newEdges,n.victim)
30:             if v = FALSE then return abort event $A_i$
31:             newEdges ← newEdges $\cup$ edgs
32:         else
33:             blinds ← blinds $\cup\{$ n $\}$
34:     ▷ Phase II — install the blind writes
35:     repeat
36:         foundOutEdges ← FALSE
37:         inEdges ← $\emptyset$
38:         for each n in blinds do
39:             ▷ find the latest possible victim
40:             (victim,edges)←FINDVICTIM(newEdges,n)
41:             if victim $=\perp$ then return abort event $A_i$
42:             for each e in edges do
43:                 if e is incoming to $T_i$ then
44:                     inEdges ← inEdges $\cup$ e
45:                 else if e $\notin$ newEdges then
46:                     newEdges ← newEdges $\cup\{$ e $\}$
47:                     foundOutEdges ← TRUE
48:     until foundOutEdges = FALSE
49:     ▷ commit point
50:     for each n in $T_i$.writeList do
51:         install the new version right after n.victim
52:     $PG$.ADDEDGES(newEdges $\cup$ inEdges)

53: procedure FINDVICTIM(List⟨Edge⟩ newEdges, WriteNode wn) : (ObjectVersion, List⟨Edge⟩)
54:     ▷ find the latest possible victim
55:     if wn.victim $=\perp$ then vctm←wn.latestVersion
56:     else vctm ← wn.victim
57:     while vctm $\neq \perp$ do
58:         ▷ check installing the new version after vctm
59:         (valid, edges)←VALIDATEWRITE(newEdges,vctm)
60:         if valid = TRUE then return (vctm,edges)
61:         vctm ← vctm.prev                                          ▷ go to the previous version
62:     return $(\perp, \perp)$                                        ▷ no suitable victim found

63: procedure VALIDATEWRITE(List⟨Edge⟩ edges, ObjectVersion $o.v_n$) : (boolean, List⟨Edge⟩)
64:     added←$\{(o.v_n.writer, T_i), (o.v_n.readers, T_i), (T_i, o.v_n.next.writer)\}$
65:     valid←acyclity of $PG$ after adding edges $\cup$ added
66:     return (valid, added)
```

Our commit operation (Algorithm 2, line 23) is divided into two phases. We call the object version after which the new version is to be installed a *victim version*. The victim version is known only for the non-blind writes (that is version, which has been read before the write, line 18). In the first phase the algorithm tries to install the non-blind writes (lines 27–33). In the second phase (lines 35–48) the algorithm tries to find the vicim versions for the blind writes in iterations. Initially, the victim is the object's latest version. In each iteration, the algorithm traverses the objects and for each one searches for the latest possible victim to install the new version without creating a cycle in PG (line 40). When victim $o.v_n$ is found, an edge from T_i to the writer of $o.v_{n+1}$ is added to PG (line 46). We add only the outgoing edges at this point, because changing the victim from $o.v_n$ to $o.v_{n-1}$ may remove some incoming edges to T_i but cannot remove outgoing ones. Meanwhile, incoming edges are kept in *inEdges*. After each iteration, there are possibly new outgoing edges added to PG, that would mean that the previously found victim versions might not suit anymore and a new iteration should be run. Once there is an iteration when no new edges are added, the algorithm commits — it installs the new versions after their victims and adds all the edges, including *inEdges* from the latest iteration, to the PG.

The following lemma immediately follows from the protocol.

Lemma 8 *Γ-AbortsAvoider maintains PG acyclic.*

Proof The edges added to the graph are defined in functions READ (line 7) and VALIDATEWRITE (line 63). Both functions validate that adding the new edges preserves PG acyclicity. □

We now want to show that the algorithm does not introduce unnecessary aborts.

Theorem 2 *Γ-AbortsAvoider forcefully aborts a transaction T in a history H only if there exists no live-T H's modification H', such that PG'_H contains no cycles.*

Proof We first note that in Γ-AbortsAvoider no transaction can abort other transactions — the only transaction that can be aborted as a result of T_i's operation invocation is T_i by itself. Hence, if Γ-AbortsAvoider aborts a set S of transactions, then $|S| = 1$. Therefore, it is sufficient to prove the following: Γ-AbortsAvoider forcefully aborts a transaction only if not aborting any transaction would create a cycle in PG.

The read operation of object o (line 4) returns A_i only if there is no object version to read without introducing a cycle in PG. The write operation (line 13) does not abort any transaction — it postpones all the work till the commit.

The commit operation (line 23) tries to write the new versions of all the objects written during the transaction. If the object is written in the non-blind way, then the victim version is known beforehand and the new version has to be installed after the version that has been read (line 29). In this case the validation is done by *validateWrite* function (line 63), which fails if and only if adding the appropriate edges to PG creates a cycle.

It remains to show that commit function does not succeed to execute the blind writes only if that creates a cycle in PG. We will show now that if there exists a way to execute the blind writes without creating a cycle in PG, the algorithm will find it.

First of all, we will analyze the variable *newEdges* (line 24), which keeps the set of the edges added to PG upon successful commit of T. Edge $(T_i, T_j) \in newEdges$ is *compulsory*, if PG must have a path from T_i to T_j after successful commit (to that end, the edge represents a real, compulsory dependency). □

Lemma 9 *During* COMMIT() *function of AbortsAvoider algorithm, the newEdges set contains compulsory edges only.*

Proof In the first phase of COMMIT(), AbortsAvoider proceeds the non-blind writes (lines 27–33). There is a single possible victim version for the non-blind write, and therefore the edges added to *newEdges* set during the first phase are compulsory.

Consider the second phase of COMMIT(), when AbortsAvoider proceeds the blind writes (lines 35–48). We will show by induction that all the edges added to *newEdges* in the second phase are compulsory.

Induction basis. At the beginning of the second phase *newEdges* set contains only the edges added by the non-blind writes, which are compulsory, as shown before.

Induction step. Assume that all the edges added to *newEdges* by the algorithm so far are compulsory. Consider new edge $(T_i, o.v_{k+1}.writer)$ added to *newEdges* by the algorithm in line 46. This happens if $o.v_k$ is chosen to be a victim version for writing to object o. According to the algorithm, $o.v_k$ is chosen to be the victim version only if all the versions $o.v_{k'}$ for $k' > k$ did not suit to be the victim versions for a given *newEdges* set. According to the induction assumption, *newEdges* set contains compulsory edges only, therefore all the versions $o.v_{k'}$ for $k' > k$ cannot be victim versions for the write operation. According to the algorithm, choosing any object version $o.v_{k'}$ for $k' \leq k$ (i.e., object version that is earlier than $o.v_k$) yields a path from T_i to $o.v_{k+1}.writer$ in PG, finishing the proof. □

For each object written in a blind way the algorithm checks the victim versions starting from the latest one. Victim version validation is executed in the following way: PG is checked for acyclity after inserting the edges from *newEdges* set together with the edges corresponding to adding the new version after $o.v_k$. As stated in Lemma 9, *newEdges* set contains compulsory edges only, therefore validation fail for $o.v_k$ means that neither $o.v_k$, nor any version later than $o.v_k$ can be the victim version of o. The algorithm traverses the objects in iterations, till it finds a combination of victim versions that does not create a cycle in PG (and then commits), or discovers object o such that none of o's versions can be the victim version (and then aborts).

Corollary 2 *Γ-AbortsAvoider satisfies Γ-opacity and online Γ-opacity-permissiveness.*

We have shown that Γ-AbortsAvoider protocol is correct and avoids unnecessary aborts. In the rest of the paper we will show the garbage collection rules and optimization techniques for the protocol.

6.3 Garbage Collection

A TM should garbage collect unused metadata. In our case, metadata consists of the objects' previous versions as well as terminated transactions. In this section, we describe how those may be garbage collected.

Read operations Consider transaction T_i reading object o. The following lemma stipulates that some of the edges added to the precedence graph in the simplified protocol are redundant, and in fact, the only edges that need to be added by the protocol during read operations are incoming ones.

Lemma 10 *When T_i reads $o.v_n$, it suffices to add one edge from $o.v_n.writer$ to T_i in PG.*

Proof We say that adding an edge (v_1, v_2) is *unnecessary*, if PG already contains a path from v_1 to v_2, thus adding this edge does not influence on the cycle detection. We will show that adding the outgoing edge from T_i to $o.v_n.writer$ during a read is unnecessary. Therefore the only edge that need to be added by the protocol is the edge from $o.v_{n-1}.writer$ to T_i.

The protocol adds outgoing edge from T_i to $o.v_n.writer$ if T_i reads version $o.v_{n-1}$. According to the algorithm, T_i tries first to read the latest version $o.v_{n+k}$, if this read creates a cycle, it tries to read $o.v_{n+k-1}$, $o.v_{n+k-2}$ and so on till it arrives to $o.v_{n-1}$. Note, that before starting the read, the graph PG was acyclic. If T_i does not succeed to read $o.v_{n+k}$, it means that adding an edge from $o.v_{n+k}.writer$ to T_i would create a cycle, hence there is a path from T_i to $o.v_{n+k}.writer$ before the start of the read. When T_i tries to read $o.v_{n+k-1}$ and does not succeed, it means that adding the edges $\{(o.v_{n+k-1}.writer, T_i), (T_i, o.v_{n+k}.writer)\}$ creates a cycle in PG. As we have concluded, before the read, PG contained a path from T_i to $o.v_{n+k}.writer$ and was acyclic, therefore adding the single edge $(o.v_{n+k-1}.writer, T_i)$ creates a cycle in PG, i.e. there was a path from T_i to $o.v_{n+k-1}.writer$ before the read. Continuing in the same way, we conclude that before the read there was a path from T_i to $o.v_n.writer$. Therefore, adding an edge from T_i to $o.v_n.writer$ is unnecessary. □

Using the optimization above, no incoming edge is ever added to a terminated transaction as a result of a read operation.

Write operations We would like to know whether the new incoming edges may be added to a terminated transaction as a result of write operation. Consider committed transaction T_i that has written to o. If the new version $o.v_n$ has been written in a non-blind way (i.e. transaction T_i has read the version $o.v_{n-1}$ and then installed $o.v_n$), then no other transaction T_j will be able to install a new version between $o.v_{n-1}$ and $o.v_n$, for that would cause a cycle between T_i and T_j. Blind writes, however, are more problematic. Consider, for example, the scenario depicted in Fig. 7. At time t_0, T_1 has no incoming edges, but we are still not allowed to garbage collect it as we now explain. There is a transaction T_2 that read object o_1 with an active preceding transaction T_3. At the time of T_3's commit, it discovers that it cannot install the last version of o_1, and tries to install the preceding version. Had we removed T_1 from PG, this would have caused a consistency violation, because we would miss the cycle between T_1 and T_3.

The example above demonstrates the importance of knowing that from some point onward, T_i may have no new incoming edges. The lemma below shows that some edge additions can be saved:

Lemma 11 *If T_i is a terminated transaction, then no incoming edges need to be added to T_i in PG as long as for each $o.v_n$ written blindly by T_i there is no reader with an active preceding transaction.*

Proof Consider a terminated transaction T_i satisfying conditions of the lemma. According to Lemma 10 no transaction may add incoming edges to T_i as a result of read operation. It remains to check the writes. According to the protocol, the incoming edge to T_i may be added only if transaction T_j installs the version prior to the version $o.v_n$ written by T_i. First of all we should notice that $o.v_n$ should be written in a blind way in order to make this scenario happen. Secondly, if T_j tries to insert a new version before $o.v_n$, it means that T_j failed to insert its version after $o.v_n$, i.e. adding the edges from T_i and from the readers of $o.v_n$ to T_j created a cycle. But we know that T_j cannot precede the readers of $o.v_n$ according to the condition of the lemma, that is why there was a path from T_j to T_i before the write operation of T_j. Therefore there is no need to add the edge from T_j to T_i when installing the new version. □

Garbage collection conditions We say that a transaction is *stabilized* if no incoming edges may be added to it in the future. At the moment when T_i has no incoming edges and it is stabilized, we know that T_i will not participate in any cycle, and thus may be garbage collected.

Theorem 3 *The terminated transaction T_i is stabilized at time t_0 if either (1) T_i has not written blindly any object version $o.v_n$, or (2) all active transactions at time t_0 and all the transactions beginning after t_0 follow T_i according to Γ.*

Proof According to Lemma 11, no incoming edges need to be added to terminated T_i in PG if T_i has no blind writes. If transaction T_j follows T_i according to Γ, then according to AbortsAvoider algorithm, PG will contain a path from T_i to T_j after `START()` operation of T_j . Therefore, T_j may not add incoming edge to T_i if $T_i \prec_\Gamma T_j$. Hence, if all active transactions at t_0 and all the transactions beginning after t_0 follow T_i according to Γ, then no new incoming edges will be added to T_i. □

For this, we deduce that terminated transactions with no incoming edges satisfying one of the conditions of Theorem 3 may be garbage collected. Note that in the runs with no blind writes, every terminated transaction is stabilized and thus the transaction may be garbage collected at the moment it has no incoming edges.

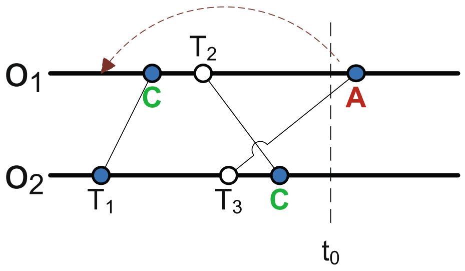


Fig. 7 The blind write of transaction T_1 does not allow us to garbage collect it at time t_0

6.4 Path Shortening and Runtime Analysis

AbortsAvoider protocol allows adding new edges to PG only if they do not introduce cycles in PG. The straightforward cycle detection algorithm runs DFS starting from T_i, traversing a set of nodes we refer to as $ingress_i$. We now present an optimization that reduces the number of nodes in $ingress_i$.

Consider stabilized terminated T_j. The idea is to connect the ingress nodes to the egress nodes of T_j directly, thus preventing DFS from traversing T_j. This becomes possible because T_j is stabilized and thus may not have new ingress nodes, hence the egress nodes do not miss the precedence info when they lose their edges from T_j. Once a terminated transaction T_j satisfies the conditions of Lemma 11 and it can no longer have additional incoming edges, (e.g., any transaction with no blind writes), we remove all of its outgoing edges by connecting its ingress nodes directly to its egress nodes as described above, and indicate that T_j is a *sink*, i.e., cannot have outgoing edges in the future. Once a transaction is marked as a sink, any outgoing edge that should be added from it is instead added from its ingress nodes. Note that our path shortening only bypasses stabilized nodes. Had we bypassed also non-stabilized ones, we would have had to later deal with adding new incoming nodes to their egress nodes, which could require a quadratic number of operations in the number of terminated transactions. Hence, we chose not to do that.

Runtime complexity of the operations Running DFS on $ingress_i$ takes $O(V^2)$, where V is the number of transactions preceding T_i, whose nodes have not been garbage collected. In the general case, $V = \#terminated + \#active$. But if all the transactions preceding Dsc_i had no blind writes, $V = \#active$.

The read operation seeks the proper version to read in the version list. Unfortunately, the number of versions that need to be kept is limited only by the number of terminated transactions. Consider the scenario depicted in Fig. 8. Here, the only version of o_2 that may be read by T_1 is the first, all other versions are written by transactions that T_1 precedes. In order to find a latest suitable version, the read operation may use a binary search – $O(\log(\#terminated))$ versions should be checked. Adding the edges takes $O(\#active)$. So altogether, the read complexity is

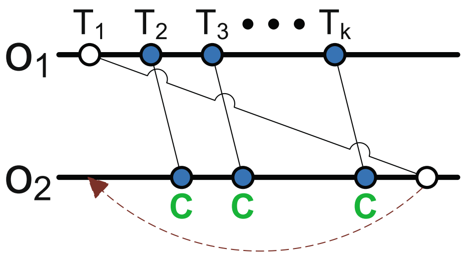


Fig. 8 All object versions must be kept, as their writers have an active preceding transaction T_2

$O(\log(\#terminated) \cdot \max\{\#active^2, \#terminated^2\})$, and $O(\log(\#terminated) \cdot \#active^2)$ when there are no blind writes.

The write operation postpones all the work till the commit. The number of iterations in the commit phase is $O(\#writes \cdot \#terminated)$, and in each iteration $O(\#writes)$ validate operations should be run. So the overall write cost is $O(\#writes^2 \cdot \#terminated \cdot \max\{\#active^2, \#terminated^2\})$, and $O(\#active^2)$ when there are no blind writes.

Finally, we would like to emphasize that although in the worst-case, these costs may seem high, transactions without blind writes are garbage collected immediately upon commit. Moreover, the only nodes in $ingress_i$ where cycles are checked are transactions that conflict with T_i. Typically, in practice, the number of such conflicts is low, suggesting that our algorithm's common-case complexity is expected to be good. On the other hand, if the number of conflicts is high, then most TMs existing today would abort one of the transactions in each of these cases, which is not necessarily a better alternative.

7 Conclusions

The paper took a step towards providing a theory for understanding TM aborts, by investigating what kinds of spare aborts can or cannot be eliminated, and what kinds can or cannot be avoided efficiently. We have shown that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it reduces the number of spare aborts: while strict online opacity-permissiveness is NP-hard, we presented a polynomial time algorithm AbortsAvoider, satisfying the weaker online opacity-permissiveness property. Understanding the properties of spare aborts is still far from being complete. For example, relaxations of the online opacity-permissiveness property or restrictions of the workload may be amenable to more efficient solutions. Moreover, the implications of the inherent "spare aborts versus time complexity" tradeoff we have shown are yet to be studied.

Acknowledgments We would like to thank Hagit Attiya, Eshcar Hillel, Alessia Minali and Rui Fan, as well as the anonymous reviewers for their helpful comments. This work was partially supported by the Israeli Ministry of Science Knowledge Center on Chip Multiprocessors and Hasso Plattner Institute.

References

1. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. In: PODC '06: Proceedings of the 25th annual ACM symposium on principles of distributed computing, pp. 308–315. ACM, New York (2006)
2. Aydonat, U., Abdelrahman, T.: Serializability of transactions in software transactional memory. In: 2nd ACM SIGPLAN workshop on transactional computing (2008)
3. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th international symposium on distributed computing, pp. 194–208 (2006)
4. Ennals, R.: Cache sensitive software transactional memory. Technical report
5. Fraser, K.: Practical lock freedom. PhD thesis. Cambridge University Computer Laboratory (2003)

6. Gramoli, V., Harmanci, D., Felber, P.: Toward a theory of input acceptance for transactional memories. Technical report (2008)
7. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Proceedings of the 22th international symposium on distributed computing (2008)
8. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, pp. 175–184 (2008)
9. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd annual symposium on principles of distributed computing, pp. 92–101 (2003)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21**(2), 289–300 (1993)
11. Keidar, I., Perelman, D.: On maintaining multiple versions in transactional memory (work in progress). Technical report, Technion (2010)
12. Moss, J.E.B.: Open nested transactions: semantics and support. In: WMPI, p. 2006, Austin
13. Napper, J., Alvisi, L.: Lock-free serializable transactions. Technical report, The University of Texas at Austin (2005)
14. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM (1979)
15. Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: Proceedings of the 26th annual ACM symposium on principles of distributed computing, pp. 340–341 (2007)
16. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 12th annual ACM symposium on principles of distributed computing (PODC), pp. 204–213 (1995)
17. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, San Mateo (2002)