On Avoiding Spare Aborts in Transactional Memory

Idit Keidar Dmitri Perelman

Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
idish@ee.technion.ac.il, dima39@tx.technion.ac.il

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 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. We also present an efficient example TM algorithm that avoids certain kinds of spare aborts, and analyze its properties and performance.

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 [11, 15] 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 multiple operations on memory objects into one transaction. Similarly to database transactions [16], 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. The Achilles' heel of most existing TMs is the fact that they 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 refrains from spare aborts.

Previous works introduced two related notions: commit-abort ratio [7] and permissiveness [8]. The latter stipulates that in runs that does not violate correctness criterion, 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 then 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. Unline ealier notions, strict online permissiveness does not prevent the TM 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 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* [13] 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.

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., [10, 6, 5, 4] use *two-phase locking*, which aborts 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. [14] 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.

Gramoli et al. [7] 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 [10]. 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 every TM is $\Omega(L)$ competitive in terms of commit-abort ratio, where L is the number of live transactions in the system. 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. [7] — a TM *accepts* a certain input pattern if it commits all of its transactions. The authors compared different TMs according to their input acceptance patterns. Guerraoui et al. [8] 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, and our correctness criterion is a generalization of *opacity* [9], 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

Transactions. Our definition of Transactional Memory (TM) is based on [9]. A TM allows threads to run *transactions*. Transactions perform operations on shared objects. The objects considered in this paper are *read/write registers*. The status of a transaction may be either *live*, *aborted*, or *committed*. A transaction can perform operations as long as it is live. Each transaction has a unique identifier (id). Retrying an aborted transaction is interpreted as creating a new transaction with a new id. The maximal possible number of live transactions is L.

The API of the TM includes the following operations. The operation ***startTransaction()*** returns the id of a newly created transacton. The status of a newly created transaction is always live. When T_i is live, it can invoke the following operations: ***read***(T_i,o), which returns the value of register o, and ***write***(T_i,o,v), which writes value v to register o. When T_i wishes to terminate, it invokes operation ***tryCommit***(T_i) or ***tryAbort***(T_i). If *tryCommit*(T_i) returns C_i, the status of T_i changes to committed, while *tryAbort*(T_i) always returns A_i, indicating that T_i is aborted. The abort value A_i may also be returned as a response to *read*, *write* or *tryCommit* invocations, in which case we say that the TM *forcefully aborts* transaction T_i. If the TM forcefully aborts transaction T_j as a result of another transaction's operation, then the returned value of the subsequent operation of T_j will be A_j. The read-set and the write-set of T_i are denoted as *read*(T_i) and *write*(T_i) respectively, and are not known in advance.

The calls to the TM are blocking — the invoking thread waits for a response before invoking more operations. We assume that TM operations issued by different threads are executed sequentially. This allows to us neglect issues related to overlapping operation executions, which are not the focus of this paper; in practice, such sequential executions can be implemented using locks for blocking implementations, or well-known non-blocking solutions, e.g., [6]. Note, however, that transactions may overlap.

The TM guarantees that each operation invocation eventually gets a response, even if all other threads are sleeping. This limits the TM'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 operations.

Transaction histories. A *transaction history* is the sequence of operations issued by transactions in a given TM execution, ordered by the time at which they are issued. Two histories H_1 and H_2 are *equivalent* if they contain the same transactions and each transaction T_i issues the same operations with the same responses in both. A history H is *complete* if it does not contain live transactions. If history H is not complete, we may build from it a complete history *Complete(H)* by adding an abort operation for every live transaction. History H' is an *extension* of H for a given TM, if (1) H is a prefix of H' and (2) H' is a possible history in the given TM. 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 event of transaction T_i is issued after the last response of transaction T_j in H, then $T_j \prec_H T_i$. 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.

Correctness. Our correctness criterion generalizes the *opacity* condition of Guerraoui and Kapalka [9]. Let $\Gamma(H)$ be a partial order on transactions. A TM satisfies Γ-*opacity* if for every history H generated by the TM there exists a sequential history S, s.t.:

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

When $\Gamma(H)$ includes all the ordered pairs of non-concurrent transactions in H, the above definition boils down to *opacity*. The use of Γ makes it possible to require the order for some subset of transactions according to any arbitrary rule; e.g., Riegel et al. [14] 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.

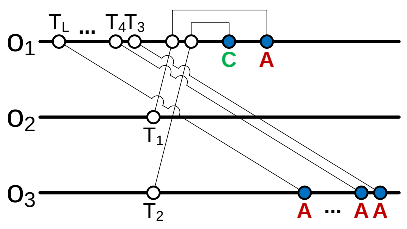
4 Limitations of Previous Measures

4.1 Commit-Abort Ratio

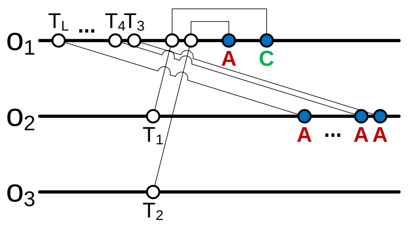
The *commit-abort ratio* (τ) [7] is the ratio between the number of committed transactions and the overall number of transactions in the history. Unfortunately, no online TM may guarantee commit-abort ratio optimally. Recall that L is the number of live transactions. We show that every TM is $\Omega(L)$ competitive in terms of its commit-abort ratio.

We use the style of [14] 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 arc indicates the point in time at which the access to the object appears according to the TM serialization.

Lemma 1. *Every TM is $\Omega(L)$ competitive in terms of its commit-abort ratio.*



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



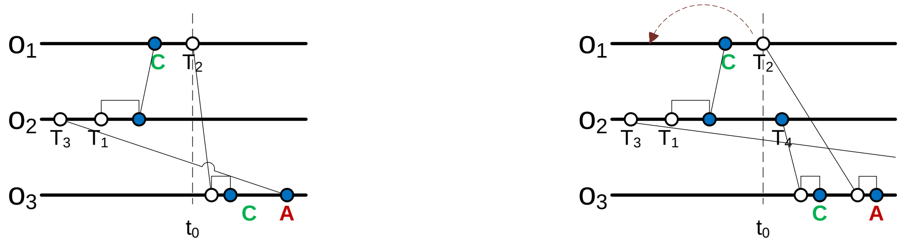
(b) Run r_2: T_1 commits, all other transactions abort: $\tau = \frac{1}{L}$

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

Proof. Consider the scenarios depicted in Figure 1. The runs are indistinguishable until the time when T_L tries to commit. Transactions T_1 and T_2 cannot both commit because both write o_1 after reading its previous value. In run r_1 (Figure 1(a)), the TM commits T_2, then T_1 aborts and then the transactions $T_3 \cdots T_L$ try to write to o_3 and must be aborted because they conflict with T_2, resulting in $\tau = \frac{1}{L}$. In run r_2 (Figure 1(b)), the TM aborts T_2, T_1 commits and then the transactions $T_3 \cdots T_L$ try to write to o_2 and therefore must be aborted, resulting again in $\tau = \frac{1}{L}$. The optimal offline TM in these cases would abort only one transaction, yielding $\tau = \frac{L-1}{L}$. The online TM, however, cannot distinguish between r_1 and r_2 at the moment it should decide whether to abort T_1 or T_2, hence the competitive ratio is $\Omega(L)$. □

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: a TM provides *permissiveness* [8] if it accepts every possible set of input patterns satisfying Γ-*opacity*. Gramoli et al. showed that existing TM implementations do not accept all inputs they could have, and hence are not permissive. We show that this is an inherent limitation.



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

Figure 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.

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

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

Proof. Consider the scenario depicted in Figure 2. All the objects have initial values, v_0. All the transactions start at the same time, t_0, and are therefore not ordered according to the real-time order, thus the third condition of our correctness criterion holds for any Γ.

T_1 writes values v_1 to o_2 and o_1. 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 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 Figure 2(a), 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 Figure 2(b). 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 will try to write to o_3 and commit, 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 TM can accept both of them. □

5 Online permissiveness: limitations and costs

5.1 Strict Online Opacity-Permissiveness

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

Definition 1. *A TM satisfies strict online Γ-opacity-permissiveness if the TM forcefully aborts transaction T_i only when not aborting any live transaction violates Γ-opacity for the given Γ.*

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 Figure 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 [13].

Given history H, the *augmented* history $\bar{H}$ is the history, which 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 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, then T_i is also live in H. Note that aborted transaction cannot be live according to this definition (no transaction may read the values written by the aborted one). 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'. History H is *view serializable*, if for any prefix H' of H, *complete(H')* is view equivalent to some serial history S.

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 will show a reduction from the NP-complete problem of detecting view-seializability of history H without dead transactions to the problem of detecting whether some history H' satisfies Γ-opacity. Consider history H with no dead transactions. In the absence of aborted transactions, the definition of view serializability 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 serializable if and only if H' satisfies Γ-opacity.

H is view serializable if and only if there exists a legal sequential history S, which is view equivalent to *Complete*(H). All the transactions in H' are concurrent (*start(T_i)* follows before any other event for every

T_i), 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-serializable. □

5.2 Online Opacity-Permissiveness

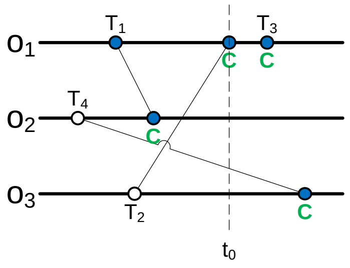


Figure 3: The order of transactions T_1 and T_2 is changed after their commit time.

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 Figure 3. Transactions T_1 and T_2 are not ordered according to real-time order, therefore Γ has no effect. 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 T_3 overwrites o_1 before any transactios reads the value written by T_2. And when T_4 commits, the serialization order becomes $T_2 \rightarrow T_4 \rightarrow T_1 \rightarrow T_3$. If the partial serialization order induced by the run cannot change after being defined, the problem becomes much easier. We capture this intuition with the following definition:

Definition 2. *A TM maintains λ-persistent ordering if it builds incrementally a partial transactional order λ according to the following rules: (1) initially λ is empty (2) at any point of time, λ must order exactly all the pairs of committed transactions T_i, T_j, s.t.* $\text{write}(T_i) \cap \text{write}(T_j) \neq \emptyset$*, (3) each time λ is updated, its new value must preserve the order previously defined by the old one.*

In other words, if T_i and T_j are committed transactions in H that have written to the same object in a given TM, then they are ordered by λ and their order will persist in every extension of the run.

We now define a more relaxed property, *online Γ-opacity-permissiveness*, which may be satisfied at a reasonable implementation cost.

Definition 3. *A TM satisfies online Γ-opacity-permissiveness for a given Γ if the TM maintains λ-persistent ordering for some λ consistent with Γ, and the TM forcefully aborts transaction T_i only when not aborting any live transaction violates $(\Gamma \cup \lambda)$-opacity.*

Note that Definition 3 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 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.

6 The AbortsAvoider Algorithm

We now present AbortsAvoider, a TM algorithm implementing online opacity-permissiveness. The basic idea behind AbortsAvoider is to maintain a precedence graph of transactions, and keep it acyclic, as explained in Section 6.1. The key challenge AbortsAvoider faces is that completed transactions cannot always be removed from the graph, whereas keeping all transactions forever is clearly impractical. We call transactions that are not live but were not removed from the graph *zombies* [1]. We address this challenge in Section 6.2, presenting a garbage collection mechanism for removing zombie transactions from the graph. In Section 6.3 we present another optimization, which shortens paths in the graph to reduce the number of zombie transactions traversed during the acyclity check. Our complexity analysis appears in the same section.

6.1 Basic Concept

Information bookkeeping. Every object is accessed via an *object handle* which points to the *version list* of the object, holding the values written to the object during the run. The object handle points to the last installed version, *o.latest* as in JVSTM [3]. Read returns some version from the list, and write creates a new version which is inserted to the list upon commit (not necessarily at the end). 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 status, a *readList* and a *writeList*. An entry in *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 updated data, and the version after which the new version should be added. Transactions may point one to each other, forming a directed graph called CG_A.

Characterization graph. A *characterization graph* CG is a directed labelled graph that reflects the dependencies between the transactions as they are created during the the run. The vertexes of CG are transactions, the edges of CG are as follows:

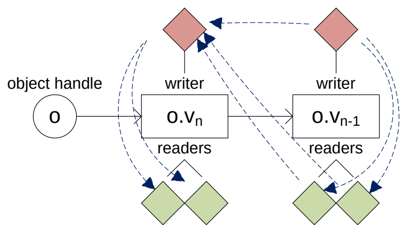


Figure 4: Object versions and the characterization graph, CG.

If $(T_j, T_i) \in \Gamma$, then CG 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 CG 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 CG 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 CG contains (T_j, T_i) labelled L_{WaR} (Write after Read).

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 the 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 [9].

[1] A zombie is a reanimated human corpse. Stories of zombies originated in the Afro-Caribbean spiritual belief system of Vodou, which told of the people being controlled as laborers by a powerful sorcerer.

Note that the characterization graph does not refer to the local operations. Below we present lemmas that link the acyclity of CG to Γ-opacity and online permissiveness.

Lemma 4. *If the CG of a run is acyclic, then the non-local history H of the run satisfies Γ-opacity.*

Proof. Let H be a history over transactions $\{T_1 \ldots T_n\}$. Let H_C be a history equal to H with the following additions: for every live transaction $T_i \in H$, we add A_i to H_C. Clearly, $H_C = \mathit{Complete}(H)$.

Since CG is acyclic, it can be topologically sorted. Let $T_{i1}, \ldots, T_{in}$ be a topological sort of CG, and let S be the sequential history $T_{i1}, \ldots, T_{in}$.

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 prove now 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. 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 the CG. 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 the CG, 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 the CG, 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, contradiction.

Finally, for each pair $(T_i, T_j) \in \Gamma$, the CG contains an edge from T_i to T_j. Therefore, according to the topological sort, S preserves the partial order Γ.

Summing up, *Complete*(H) is equivalent to a legal sequential history S, and S preserves partial order Γ. Therefore H is Γ-opaque. □

Lemma 5. *Consider a TM that forcefully aborts a transaction only if not aborting any transaction would create a cycle in the characterization graph of the run. Then this TM satisfies online opacity-permissiveness.*

Proof. We need to show a partial transactional order λ, s.t. λ-persistent ordering is maintained in the TM and there is a cycle in the CG if and only if $(\Gamma \cup \lambda)$-opacity is violated. We define λ in the following way: if two committed transactions T_i and T_j have written to the same object to the versions $o.v_n$ and $o.v_m$ respectively, where $n < m$, then $T_i < T_j$ according to λ.

We show first that λ-persistent ordering is maintained in the TM. Clearly, the initial value of λ is an empty set. Transactions T_i and T_j are ordered by λ if and only if they are both committed and $\mathit{write}(T_i) \cap \mathit{write}(T_j) \neq \emptyset$. Finally, the updates of λ preserve the order induced by the previous values of λ.

We want to show now that if there is an edge (T_i, T_j) in the CG, then any legal sequential history S preserving $\Gamma \cup \lambda$ and equivalent to *Complete(H)* should order T_i before T_j. Consider two transactions T_i and T_j s.t. there is an edge (T_i, T_j) in the CG. If the edge is labelled L_Γ, then $(T_i, T_j) \in \Gamma$, therefore S should order T_i before T_j. If the edge is labelled L_{RaW}, then T_j reads a value written by T_i and S should order T_i before T_j. If the edge is labelled L_{WaW}, then T_i and T_j are committed transactions writing to the same object, $T_i < T_j$ according to $\lambda(H)$, hence S should order T_i before T_j. If the edge is labelled L_{WaR}, then T_i reads $o.v_n$ while T_j writes $o.v_{n+1}$. From one side, T_j should be ordered after the $o.v_n.\mathit{writer}$ in S (L_{WaW} edge, see above), from the other side T_j cannot be ordered between $o.v_n.\mathit{writer}$ and T_i, because T_i must read the value written by $o.v_n.\mathit{writer}$ in S. Therefore, T_j should be ordered after T_i in S.

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

A characterization graph was previously used by Guerraoui and Kapalka [9]. However, the graph they built was a function of history H and some total order chosen on the set of transactions in H. In our case, the characterization graph depends on an algorithm's run, including its internal decisions, i.e., which object version is accessed in each operation.

Simplified Γ-AbortsAvoider Algorithm. Γ-AbortsAvoider is depicted in Algorithm 1 in Appendix A. A *read* operation looks for the latest possible object version to read without creating a cycle in CG_A. *Write* operations postpone the actual work till the commit. The *commit* operation is more complicated. Intuitively,

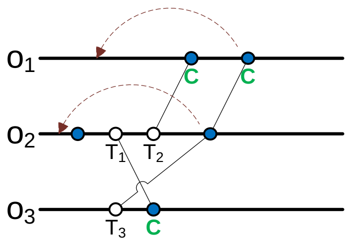


Figure 5: Checking the written objects in a greedy way during the commit may lead to a spare abort.

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 Figure 5. 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 one would have allowed T_3 to commit, hence aborting T_3 violates Γ-no-spare aborts.

Our commit operation thus works in iterations. We call the object version after which the new version is to be installed a *victim version*. Initially, the victim version of every written object is the last one. In each iteration, the algorithm traverses the written objects and for each one searches the latest possible victim to install the new version without creating a cycle in CG_A. When victim $o.v_n$ is found, an edge from T_i to the writer of $o.v_{n+1}$ is added to CG_A. 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. After each iteration, there are possibly new outgoing edges added to CG_A, 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 to the CG_A. Note that victim search is relevant only for so-called blind writes; a write that is preceded by a read inserts the new version right after the read version.

The following theorem immediately follows from the protocol.

Theorem 2. *Γ-AbortsAvoider forcefully aborts a transaction if and only if not aborting any transaction would create a cycle in CG_A.*

Proof. The function *validateGraph(newEdges)* (line 41) returns false if and only if the new edges added to CG_A create a cycle in CG_A. The read operation (line 7) of object o returns A_i if and only if reading any version of o does not pass the check of *validateGraph*, i.e., reading any version of o creates a cycle in CG_A. Write operation (line 17) cannot forcefully abort any transaction because all the work is postponed till the commit.

Commit operation (line 46) tries to write the new versions for each object written during the transaction. If the object is written in the non-blind way, then the new version must be installed directly after the version which has been read. This is checked by *validateWrite* (line 85) function which returns false if and only if adding the appropriate edges to the characterization graph creates a cycle in CG_A.

It remains to show that commit function does not succeed to execute the blind writes if and only if that would create a cycle in CG_A. The first direction is a trivial one: CG_A is checked for cycles every time we try to insert the new version, that is why if the algorithm succeeds to commit, CG_A with the addition of the new edges is still acyclic. We will show now that if there exists a way to execute the blind writes without creating a cycle in CG_A, the algorithm will find it.

First of all, we will analyze the variable *newEdges*, which keeps a set of the edges which will be added to CG_A as a result of successful commit. We say that the edge (T_i, T_j), added to *newEdges* set during the commit function is *compulsory*, if CG_A must have a path from T_i to T_j if the commit succeeds (thus, the edge represents a real, compulsory dependency).

Lemma 6. *In commit function of AbortsAvoider algorithm,* newEdges *set contains compulsory edges only.*

Proof. At the first stage of the commit, AbortsAvoider writes the non-blind writes. Non-blind write to object o has only one choice for inserting the new object version (it should go directly after the object version which has been read by T_i). Therefore, the edges added to *newEdges* set as a result of the non-blind write are compulsory.

Consider the second stage of the commit, at which AbortsAvoider executes the blind writes. The only edges added to *newEdges* at this stage are the outgoing edges from T_i. We will show by induction, that all the edges in *newEdges* are compulsory. Induction basis. At the beginning of the second stage of commit, *newEdges* set contains only the edges added as a result of the non-blind writes, which are compulsory, as shown before. Induction step. Let's assume, that all the edges added to *newEdges* by the algorithm so far are compulsory. Consider the new edge $(T_i, o.v_{k+1}.\mathit{writer})$ added to *newEdges* by the algorithm in line 67. This edge is added because the algorithm has chosen $o.v_k$ to be a victim version for writing to object o. This can happen only if all the versions $o.v_{k'}$ for $k' > k$ did not suit to be victim versions for a given *newEdges* set. But according to the induction assumption, *newEdges* set contains compulsory edges only, that is why all the versions $o.v_{k'}$ for $k' > k$ cannot be victim versions for the write operation. According the algorithm, choosing any object version $o.v_{k'}$ for $k' \leq k$ (i.e., object version which is "older" than $o.v_k$) would create a path from T_i to $o.v_{k+1}.\mathit{writer}$ in CG_A, finishing the proof. □

For each object which should be written in a blind way the algorithm seeks a victim version starting from the last one. The check of victim version $o.v_k$ is executed in the following way: the edges from *newEdges* set, and the edges which should be put as a result of adding the new version after $o.v_k$, are added to CG_A, afterwards CG_A is checked for acyclity. As stated in Lemma 6, *newEdges* set contains compulsory edges only, therefore if the check fails for the version $o.v_k$, we know that neither $o.v_k$, nor any other 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 which does not create a cycle in CG_A (and thus commits), or it arrives to the earliest version of some object o and finds out that none of the versions of object o can be the victim version (and thus aborts). □

Corollary 1. *Γ-AbortsAvoider satisfies Γ-opacity and online Γ-opacity-permissiveness.*

6.2 Reducing Edges and Garbage Collection

The simplified protocol described above processes a large amount of information in each check. Clearly, any practical TM should take care of garbage collecting unused metadata. In our case, metadata consists of the objects' previous versions together with finished transactions. In this section, we describe how those may be garbage collected. Moreover, CG_A includes many redundant edges, we now explain how these can be saved.

Read operations. Consider transaction T_i reading object o. The following lemma stipulates that some of the edges added to descriptors 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 7. *When T_i reads $o.v_n$, it suffices to add one edge from $o.v_n$.writer to T_i in CG_A.*

Proof. We say that adding an edge (v_1, v_2) is *unnecessary*, if CG_A 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 CG_A 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 CG_A. As we have concluded, before the read, CG_A 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 CG_A, 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. □

We thus modify the protocol so as not to add unnecessary edges (removing line 36 from the pseudo-code in Appendix A). Using the optimization above, no incoming edge is ever added to a zombie transaction as a result of a read operation.

Write operations. We say that a write operation to o is *"blind"* in T_i if it is not preceded by read of o in T_i. Clearly, if the new object version $o.v_n$ is not a blind write (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 Figure 6. 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 a live 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 earlier version. Had we removed T_1 from CG_A, 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 8. *If T_i is a zombie transaction, and for each $o.v_n$ written blindly by T_i there is no reader with a live preceding transaction, then no incoming edges need to be added to T_i in CG_A.*

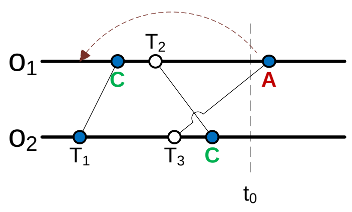


Figure 6: The blind write of transaction T_1 does not allow to garbage collect it at time t_0.

Proof. Consider a zombie transaction T_i satisfying conditions stated in the lemma. According to Lemma 7 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 some transaction T_j installs the version previous 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 can't precede none of 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 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. We thus remove from the graph every transaction T_i s.t. (1) T_i is a zombie, (2) T_i has no incoming edges, and (3) for every object version $o.v_n$ written by T_i in a blind way there is no transaction T_j that read $o.v_n$ and has a live preceding transaction. In runs with no blind writes, the third condition always holds and the transactional descriptor may be garbage collected at the moment it has no incoming edges.

When clearing the object versions, we should take into account, that the object version is needed as long as there is a possibility that some transaction will need to read its value, and as long as its readers or writer may participate in a cycle. T_i may need to read $o.v_n$ either if it is the latest version, or if T_i cannot read $o.v_{n+1}$ without creating a cycle, i.e., if there is a path from T_i to $o.v_{n+1}.\mathit{writer}$. Therefore, if $o.v_{n+1}.\mathit{writer}$ does not have incoming edges and will not have any incoming edge in the future, no transaction T_i will ever read $o.v_n$. Thus, object version $o.v_n$ may be garbage collected if (1) $o.v_n$ is not the latest one, (2) $o.v_{n+1}.\mathit{writer}$ may be garbage collected, and (3) $o.v_n.\mathit{readers}$ and $o.v_n.\mathit{writer}$ may be garbage collected.

6.3 Path Shortening and Runtime Analysis

To detect cycles when committing T_i, AbortsAvoider runs DFS starting from T_i, traversing a set of nodes we refer to as $\mathit{ingress}_i$. We now present an optimization that reduces the number of nodes in $\mathit{ingress}_i$.

Consider stabilized zombie T_j. The idea is to connect the ingress nodes to the egress nodes of T_j directly, thus preventing from DFS traversing T_j. This becomes possible because T_j is stabilized and thus may not have new ingress nodes, and the egress nodes do not miss the precedence info when they loose their edges from T_j. Once a zombie transaction T_j satisfies the conditions of Lemma 8 that 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 incoming nodes to their egress nodes, which could require a quadratic number of operations in the number of zombies. 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, $V = \textit{\#zombies} + \textit{\#live}$. But if all the transactions preceding Dsc_i had no blind writes, $V = \textit{\#live}$.

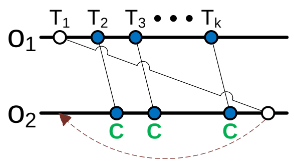


Figure 7: All object versions must be kept, as their writers have a live preceding transaction T_2.

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 zombie transactions. Consider the scenario depicted in Figure 7. 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 – $O(\log(\textit{\#zombies}))$ versions should be checked. Adding the edges takes $O(\textit{\#live})$. So altogether, the read complexity is $O(\log(\textit{\#zombies}) \cdot \max\{\textit{\#live}^2, \textit{\#zombies}^2\})$, and $O(\log(\textit{\#zombies}) \cdot \textit{\#live}^2)$ 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 $O(\textit{\#writes} \cdot \textit{\#zombies})$, and in each iteration $O(\textit{\#writes})$ validate operations should be run. So the overall write cost is $O(\textit{\#writes} \cdot \textit{\#zombies} \cdot \textit{\#live}^2)$, and $O(\textit{\#live}^2)$ when there are no blind writes.

Finally, we would like to emphasize that although in the worst-case, these costs may seem high, in the common case, where transactions do not perform blind writes, nodes 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.

References

[1] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 308–315, New York, NY, USA, 2006. ACM.

[2] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking 2. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[5] R. Ennals. Cache sensitive software transactional memory. Technical report.

[6] K. Fraser. Technical report number 579 practical lock-freedom, 2004.

[7] V. Gramoli, D. Harmanci, and P. Felber. Toward a theory of input acceptance for transactional memories. Technical report, EPFL, 2008.

[8] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in Transactional Memories. In *DISC 2008*, 2008.

[9] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, 2008.

[10] M. Herlihy and M. Moir. Software transactional memory for dynamic-sized data structures. pages 92–101. ACM Press, 2003.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[12] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues (WMPI'06)*, 2006.

[13] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 1979.

[14] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 340–341, 2007.

[15] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[16] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

A Pseudo-code

Algorithm 1 Simplified Γ-AbortsAvoider algorithm for transaction T_i.

```
procedure START()
  status ← live
  readList.reset()
  writeList.reset()
  $T_{prev}$ ← the latest transaction preceding $T_i$ by partial order $\Gamma$
  addEdges($\{(T_{prev}, T_i)\}$)

procedure READ($o$)
  if $o \in T_i$.writeList then return $T_i$.writeList[$o$].data
  if $o \in T_i$.readList then return $T_i$.readList[$o$].data
  (curVersion,newEdges) ← version2Read($o$)
  if curVersion = NULL then return abort event $A_i$
  addEdges(newEdges)
  readNode ← emptyReadNode()
  readNode.version ← curVersion
  $T_i$.readList.insert(readNode)
  return curVersion.data

procedure WRITE($o, v$)
  if $o \in T_i$.writeList then
    $T_i$.writeList[$o$].data ← $v$
    return
  writeNode ← emptyWriteNode()
  writeNode.object ← $o$
  if $o \in T_i$.readList then
    writeNode.ver2Insert ← $T_i$.readList[$o$].version
  else
    writeNode.ver2Insert ← NULL
  writeNode.data ← $v$
  $T_i$.writeList.insert(writeNode)

procedure VERSION2READ($o$) : (version, edges)
  curVersion ← $o$.latest
  while curVersion ≠ NULL do
    // add the edges, check the graph
    addedEdges ← $\emptyset$
    addedEdges ← addedEdges ∪ {(curVersion.writer, $\mathrm{Dsc_i}$)}
    if curVersion.next ≠ NULL then
      addedEdges ← addedEdges ∪ {($\mathrm{Dsc_i}$, curVersion.next.writer)}
    if validateGraph(addedEdges) = TRUE then
      return (curVersion, addedEdges)
    curVersion ← curVersion.prev
  return (NULL,NULL)

procedure VALIDATEGRAPH(newEdges) : boolean
  // Look for a cycle in the graph with added newEdges
  // Run DFS on the ingress of the $T_i$,
  // while moving backward on the precedence edges.
  // Return false if cycle found, true otherwise

procedure COMMIT
  newEdges ← $\emptyset$
  blindWrites ← $\emptyset$
  //install the writes which are not blind
  for each node $o$ in $T_i$.writeList do
    if $o$.ver2Insert ≠ NULL then
      (valid,addedEdges) ← validateWrite(newEdges, $o$)
      if valid = FALSE then return abort event $A_i$
      newEdges ← newEdges ∪ addedEdges
    else
      blindWrites ← blindWrites ∪ $o$
  //install the blind writes
  repeat
    newOutgoing ← FALSE
    allNewEdges ← newEdges
    for each node $o$ in blindWrites do
      (valid,addedEdges) ← version2Write(newEdges, $o$)
      if valid = FALSE then return abort event $A_i$
      for each edge $e$ in addedEdges do
        allNewEdges ← allNewEdges ∪ $e$
        if $e$ is outgoing from $T_i \wedge e \notin$ newEdges then
          newEdges ← $e$
          newOutgoing ← TRUE
  until newOutgoing = FALSE
  //commit point
  for each node $o$ in writList do
    installVersion(o)
    addEdges(allNewEdges)

procedure VERSION2WRITE(newEdges,$o$,$v$) : (version,edges)
  if v = NULL then
    curVersion ← $o$.latest
  else
    curVersion ← v
  while curVersion ≠ NULL do
    //add the edges, check the graph
    (valid, addedEdges) ← validateWrite(newEdges, curVersion)
    if valid = TRUE then return (curVersion, addedEdges)
    curVersion ← curVersion.prev
  return (NULL,NULL)

procedure VALIDATEWRITE(newEdges, version) : (boolean, edges)
  // add the edges, check the graph
  addedEdges ← $\emptyset$
  addedEdges ← addedEdges ∪ {(version.writer, $\mathrm{T_i}$)}
  for each reader ∈ version.readers do
    addedEdges ← addedEdges ∪ {(reader, $\mathrm{T_i}$)}
  if version.next ≠ NULL then
    addedEdges ← addedEdges ∪ {($\mathrm{T_i}$, version.next.writer)}
  if validateGraph(newEdges ∪ addedEdges) = TRUE then
    return (TRUE, addedEdges)
  return (FALSE, NULL)
```