How Live Can a Transactional Memory Be?[*]

Rachid Guerraoui Michał Kapałka

February 18, 2009

Abstract

This paper asks how much liveness a transactional memory (TM) implementation can guarantee. We first devise a formal framework for reasoning about liveness properties of TMs. Then, we prove that the strongest liveness property that a TM can ensure in an asynchronous system with transaction crashes is a property that we call *global progress*. This property is analogous to lock-freedom for shared-memory objects and is indeed guaranteed by certain TM implementations, e.g., OSTM [7]. We also prove that the presence of *zombie* transactions, which perform infinitely many operations but never attempt to commit, does not impact our result. In fact, we show that zombie transactions are, in a precise sense, equivalent to crashed transactions.

1 Introduction

Transactional memory (TM) [14, 20] is a promising paradigm that aims at bringing efficient concurrent programming to non-expert programmers. Basically, a TM allows processes (threads) of an application to communicate by executing lightweight, in-memory transactions. Each transaction accesses shared data and then either commits or aborts. When it commits, the transaction appears to the application as if all its operations were executed *atomically*, at some single and unique point in time. When it aborts, however, all the changes done to the shared state by the transaction are rolled back and are never visible to other transactions. The TM paradigm is considered as easy to use as coarse-grained locking, and has some potential for exploiting the underlying multi-core architectures as efficiently as handcrafted, fine-grained locking that is often an engineering challenge. It is thus not very surprising to see a large body of work dedicated to implementing the TM paradigm and reducing its overheads.

Some recent work has also been devoted to formally defining the semantics of TM and determining the inherent limitations of the paradigm. For example, a correctness condition for TMs has been proposed in [9], and the programming language level semantics of specific classes of TM implementations has been given, e.g., in [22, 15, 1, 18, 17]. Also, the progress semantics of two classes of TM implementations, obstruction-free and lock-based TMs, has been formalized in [8, 10]. All those papers, however, focus on *safety*.[1]

In this paper, we ask the question of how much *liveness* a TM implementation can guarantee, and make a first step towards answering that question. We consider here two kinds of transaction failures: *crashed* and *zombie* transactions. Ensuring liveness despite crashes models the requirement that a long delay of a transaction, due to preemption, page faults, or I/O, should not hamper the progress of other, concurrent transactions. Zombie transactions, which keep executing operations but never attempt to commit, can result from bugs in the user application (e.g., infinite loops) or even malicious behavior of some processes. Ensuring liveness despite their existence seems also very desirable.

At first sight, the question of how live a TM can be might seem trivial when there are no zombie transactions—one could think of treating an entire transaction as a single operation on some shared object and then apply the classical formalism of the wait-free hierarchy of objects [12]. Then, for example, implementing even the very strong liveness property such as wait-freedom [12] would be possible in a system that provides shared objects with sufficient power, e.g., the now ubiquitous compare-and-swap. This is, however, a wrong abstraction: transactions are not black-box operations on some shared object, because the result of every operation a transaction executes is visible to the user application. Indeed, we show that some liveness properties (such as wait-freedom) cannot be ensured by any TM in an asynchronous system, regardless of the objects used by the TM implementation.

We first focus on asynchronous systems in which all transaction failures are due to crashes, and we prove that, roughly speaking, the strongest liveness property that can be ensured by a TM in such a system is *global progress*—a property analogous to lock-freedom for shared-memory objects. There indeed exist nonblocking TM implementations that ensure global progress, e.g., OSTM [7], which thus can be thought of as optimal with respect to liveness.

To prove the result, we first identify a property (of a TM liveness property) that we call $(n-1)$-*prioritization* (where n is the number of processes of an application,

[*]EPFL Technical Report LPD-REPORT-2009-001. Submitted for publication.

[1]The progress properties defined in [8, 10] are, in fact, also safety (not liveness) properties.

i.e., the maximum number of transactions that can be concurrent at any time), and show that every TM liveness property that is *not* $(n-1)$-prioritizing is *weaker* than global progress, which, as we show, can be ensured in an asynchronous system with transaction crashes. We also prove that every TM liveness property that is *nonblocking* and $(n-1)$-prioritizing is impossible to implement in such a system. (Intuitively, a TM liveness property is nonblocking if it ensures progress of any transaction that runs alone, i.e., without any concurrent non-faulty transactions.) The $(n-1)$-prioritization property is thus a necessary and, within the class of nonblocking TM liveness properties, sufficient condition for a TM liveness property to be impossible to implement in an asynchronous system with transaction crashes—a result interesting in its own right.

Intuitively, a TM liveness property is $(n-1)$-prioritizing if it specifies, for some execution involving infinitely many transactions, a set of $n-1$ transactions that have *high priority*. The property then requires that in this specific execution at least one of the high-priority transactions must commit. The reason why every nonblocking, $(n-1)$-prioritizing TM liveness property is impossible to implement is intuitively the following. Any process that executes infinitely many transactions can prevent all the other $n-1$ processes from committing *any* transaction. For example, if n concurrent transactions first read a value v from a shared variable x and then they all write to x a value different than v, then only one of those transactions can commit—otherwise, the safety property of the TM (opacity [9] or even serializability [19, 4]) would be violated. Hence, if at most $n-1$ transactions have high priority, they can all be prevented from committing.

We then ask whether the presence of zombie transactions change, in terms of computability, the ability of a TM to guarantee a given liveness property. Maybe surprisingly, the answer is "no". First, we prove that global progress can still be ensured even if transactions can be zombies. (In fact, OSTM already tolerates zombie transactions, although this has not been formally shown.) Second, we show that every TM implementation M that ensures some *well-formed* liveness property L and tolerates zombie transactions, but does not tolerate transaction crashes, can be transformed into a TM implementation M' that ensures L while tolerating both zombie and crashed transactions. (Roughly speaking, a liveness property is well-formed if it does not guarantee more when there is more concurrency between transactions. Every liveness property ensured by existing TMs is, in fact, well-formed.) Intuitively, the transformation from M to M' basically "changes" crashed transactions into zombies by inserting periodically into every pending transaction some operations that cannot be observed by the user application. Those transactions are then executed by M. The major technical difficulty here has to do with the fact that there is no way to force a crashed transaction to keep executing those inserted operations. We circumvent the problem by replicating M onto each process, and make this process execute transactions of all the processes of the application on its local replica of M.[2] Then, even if some process (and its transaction) crashes, the other processes can still make the crashed transaction execute operations. (Of course, the replicas at non-crashed processes have to be kept in sync.)

To prove these results, we needed first to devise a theoretical framework for reasoning about the liveness of a TM implementation. In particular, we defined what a TM liveness property is precisely, in a way that (1) abstracts only the key aspects of the liveness semantics of a TM, thus simplifying the formal reasoning about TM liveness properties, and (2) allows describing the liveness guarantees of existing TMs. We believe the framework to be an important contribution in its own right.

To summarize, this paper addresses the question of how much liveness a TM implementation can ensure in an asynchronous system. We define precisely the notion of a TM liveness property, prove that the strongest (nonblocking) TM liveness property that can be ensured in an asynchronous system with transaction crashes is global progress (a property of some existing TMs), and show that considering zombie transactions does not change in a fundamental way the implementability of TM liveness properties.

Roadmap. The rest of the paper is organized as follows. Section 2 defines our TM system model. Section 3 defines the notion of a TM liveness property and gives several examples of TM liveness properties ensured by existing TM implementations. Section 4 determines the strongest (nonblocking) TM liveness property that can be ensured in an asynchronous system with transaction crashes. Section 5 discusses zombie transactions. Finally, Section 6 discusses our results and provides a number of open questions.

2 Model

Processes and transactions. We assume an asynchronous, shared memory system of *n processes* $p_1, \ldots, p_n$ that communicate by executing *transactions*. Each transaction has a unique *transaction identifier* from infinite set $\mathcal{T} = \{T_1, T_2, \ldots\}$. We say that a transaction T_i (i.e., a transaction with identifier T_i) performs an action, meaning that some process p_k performs this action within the transactional context of T_i.

Each transaction T_i may perform any number of operations on *transactional objects* (or *t-objects*, for short). Transaction T_i may also issue special operations: $tryC(T_i)$ and $tryA(T_i)$. Operation $tryC(T_i)$ is a request to commit T_i. When *tryC* returns value C_i, this means that T_i is *committed*. Once a transaction is committed, it can no longer perform any action. Operation $tryC(T_i)$ can also return value

[2]The idea here is similar to that behind the universal construction [12].

A_i, which means that T_i has been *aborted*. An aborted transaction continues its execution (after a roll-back) and may eventually commit. Operation $tryA(T_i)$, which always returns value A_i, is a request to abort transaction T_i. In fact, we assume that every operation of a transaction T_i can return the special value A_i. If T_i is aborted, T_i is restarted by the TM.[3]

An *event* is any invocation or response of an operation issued by any transaction (i.e., by the process executing this transaction). A response event that returns value C_k or A_k (for any k) is called, respectively, a *commit event* and *abort event*.

A *TM implementation* is any algorithm that implements the operations issued by transactions, using a number of *base objects* (e.g., provided in hardware). This algorithm is executed by the same processes that execute transactions. We call the operations executed by processes on base objects *steps*.

Histories. Let M be any TM implementation. A *history* (of M) is a sequence of all (1) events that were issued on, or received from, M by all transactions, and (2) steps of M executed by transactions (processes), in a given run (of an application/system). We assume here that every event or step e can be assigned a point in time when e was executed, and that all events and steps (in a given run) can be totally ordered according to their execution time. (If several events or steps are executed at the same time, e.g., on multi-processor systems, they can be ordered arbitrarily.)

Let H be any history. We denote by $H|p_i$ and $H|T_k$ the longest subsequence of H that contains only events and steps of, respectively, process p_i and transaction T_k. We say that a transaction T_k is in H, and write $T_k \in H$, if $H|T_k$ is a non-empty sequence. We say that a transaction $T_k \in H$ is *committed* (in H), if $H|T_k$ contains operation $tryC(T_k)$ returning value C_k. We say that a transaction $T_k \in H$ is *pending* (in H), if T_k is not committed in H.

Let T_i and T_k be any two transactions in H. We say that T_i *precedes* T_k (in H), if T_i is committed and the last event of T_i precedes the first event of T_k. If neither T_i precedes T_k, nor T_k precedes T_i, then we say that T_i and T_k are *concurrent* (in H). If Q is a subset of the set of transactions in H, then we denote by $Committed_H(Q)$ the subset of those transactions in Q that are committed in H.

We assume that every history H is *well-formed*: (1) every transaction $T_k \in H$ is executed only by a single process (i.e., $(H|p_i)|T_k$ is non-empty only for one process p_i), (2) no two transactions executed in H by the same process are concurrent, and (3) if a transaction $T_k \in H$ is committed, then no event follows operation $tryC(T_k)$ returning C_k in $H|T_k$.

We denote by $p(T_k)$ the process that executes T_k in H, i.e., a process p_i such that $(H|p_i)|T_k = H|T_k$.

Let T_i be any transaction in H and t be any time. We say that T_i *has started by* t (in H) if the first event of T_i in H is executed before time t. We say that T_i is *pending at* t if (1) T_i is started by t, and (2) T_i either is pending in H or the response event of operation $tryC(T_i)$ that returns value C_i is executed after t.

Sub-transactions. Let H be any history of a TM implementation M and T_k be any transaction in H. We divide T_k into one or more *sub-transactions*,[4] denoted by $T_k^1, \ldots, T_k^m$, such that every sub-history $H|T_k^1, \ldots, H|T_k^{m-1}$ ends with a response event of an operation that returns value A_k. That is, every sub-transaction of T_k, except possibly T_k^m, is itself an aborted transaction. Basically, sub-transactions of T_k represent the subsequent retries of the computation of T_k after aborts of T_k. Note, however, that each sub-transaction of T_k may perform different operations on different t-objects.

Correctness condition. We assume that every TM implementation M ensures *opacity* [9]. Intuitively, this means that in every history H of M, every sub-transaction (of any transaction) appears as if it was executed at some single, unique point in time between its first and its last event. In particular, this means that every sub-transaction in H (even an aborted one) observes a consistent state of the system and does not observe any changes done by any aborted sub-transaction.

Crashes and zombie transactions. A system is *crash-prone* if any process in this system can, at any time, fail by *crashing*. Once a process p_i crashes, p_i does not perform any further actions. A system in which no process ever crashes is called *crash-free*.

We assume that a process p_i that does not crash keeps executing steps forever (those can be, e.g., no-ops when p_i does not execute any transaction). Hence, if H is any infinite history, then $H|p_i$ is infinite for every process that does not crash—we say then that p_i is *correct* in H.

For every infinite history H (of any TM implementation) we specify a set $Z(H)$ of *zombie* transactions. If a transaction $T_i \in H$ is a zombie transaction, and T_i is not blocked inside any operation infinitely long, then T_i executes infinitely many operations in H but never invokes operation $tryC(T_i)$. Hence, a zombie transaction can never be committed in any history. We assume that the number of operations a single sub-transaction can execute is not bounded. That is, a transaction can be detected as a zombie only when it executes infinitely many operations.

Let H be any infinite history. We say that a transaction $T_k \in H$ is *correct* in H if either (1) T_k is committed, or (2) sub-history $H|T_k$ is infinite (i.e., process $p(T_k)$ is correct in H) and T_k is not a zombie. A transaction that is not correct is called *faulty*. We assume that a correct transaction T_k can invoke operation $tryA(T_k)$ infinitely many times only if T_k is returned A_k from infinitely many operations different than $tryA(T_k)$.

Let Q be any subset of the set of transactions in history

[3]In our model, to avoid the restart of an aborted transaction T_i, the application may simply invoke $tryC(T_i)$ immediately after T_i aborts, i.e., commit an empty transaction T_i that modifies no t-objects.

[4]Sub-transactions are not nested transactions. For simplicity, we do not consider nesting of transactions within our model.

H. We denote by $Correct_H(Q)$ the subset of those transactions in Q that are correct in H.

3 TM Liveness

Intuitively, a TM liveness property describes which of the transactions in a history H have to (eventually) commit. A base of our formal definition of a TM liveness property are the following intuitive requirements, which should be satisfied by every TM liveness property L: First, L should be indeed a *liveness* property: L can be violated only in infinite histories, and only by transactions that are pending in those histories. In particular, any history in which all transactions are committed must ensure L. Hence, we require that (roughly speaking) if a history H ensures L, then every suffix of H also ensures L. Second, L can only restrict *correct* transactions: a faulty transaction T_i may perform too few steps to reach its commit phase, or cease to invoke operation $tryC(T_i)$ (in which case the TM implementation cannot commit T_i). Third, L should not depend on the exact interleaving of steps of concurrent transactions: indeed, L should not depend on something that an application that uses a TM (or even the TM implementation itself) has little influence on. We also focus in this paper on properties that are *not* functions of the sets of t-objects accessed by transactions and, in particular, of the *conflicts* between transactions. That is, we want to provide certain liveness guarantees regardless of what computations are performed by transactions. This is important because conflicts are virtually unavoidable, especially *false* conflicts, which are caused by the internal mechanisms of a TM and thus are not directly visible outside of a TM implementation.

In this section, we give a definition of the notion of a TM liveness property and illustrate it with a series of examples.

3.1 Definition of TM Liveness

Let H be any history and t be any time. Let t' be the nearest time after t (if any) at which no transaction is pending. (Time t' can be thought of as the next quiescence time after t.) Consider the set Q of transactions in H that are pending at some time between t' and t. Observe first that all transactions in Q are directly or indirectly concurrent (we formalize this notion in the next paragraphs). Roughly speaking, a TM liveness property L specifies, for every such set Q of concurrent transactions, possible subsets of the *correct* transactions from set Q that have to be committed in history H. If those transactions (and possibly other ones) are indeed committed in H, then H ensures L. More formally:

Definition 1 *A* TM liveness *property L is any function $L : 2^{\mathcal{T}} \mapsto 2^{2^{\mathcal{T}}}$ such that $S \subseteq C$ for every set $S \in L(C)$.*

Let H be any history and t be any point in time. We denote by $Concurr_H(t)$ the (minimal) set C of transactions defined recursively in the following way: (1) if a transaction $T_i \in H$ is pending at t, then $T_i \in Concurr_H(t)$, and (2) if a transaction $T_k \in H$ is pending after t and is concurrent to some transaction in $Concurr_H(t)$, then $T_k \in Concurr_H(t)$.

Definition 2 *A history H* ensures *a TM liveness property L if, for every time t, if $Q = Concurr_H(t)$ and $C = Correct_H(Q)$ then $Committed_H(C) \supseteq S$ for some set $S \in L(C)$.*

Definition 3 *A TM implementation M ensures a TM liveness property L, if every history H of M ensures L.*

Definition 4 *Let L and L' be any two TM liveness properties. We say that L is* weaker *than L' if every history that ensures L' also ensures L.*

3.2 Examples of TM Liveness Properties

We give here examples of common TM liveness properties.

Total progress. Intuitively, a TM implementation M ensures *total progress* (analogous to *wait-freedom* for shared-memory objects, when considered in a crash-prone system), if in every infinite history of M every correct transaction eventually commits. More formally, total progress is the function $L_{\Diamond\mathrm{p}}(C) = \{C\}$. It is worth noting that every TM liveness property is weaker than $L_{\Diamond\mathrm{p}}$.

Implementing a TM that guarantees total progress in a crash-prone system is, in general, impossible (we prove it in Section 4). However, if crashes of transactions can be (eventually) detected, one can ensure total progress (with no zombie transactions) by, e.g., combining an obstruction-free TM implementation (e.g., DSTM [13]) with a wait-free contention manager [11].

Ensuring total progress in a crash-free system without zombie transactions is possible (e.g., a simple TM that synchronizes all transactions using a single global lock and thus never aborts a transaction). However, none of the major existing TM implementations ensures total progress.

Theorem 5 *An infinite history H ensures $L_{\Diamond\mathrm{p}}$ if, and only if, every correct transaction in H is committed in H.*

Proof. ($\Rightarrow$) Let H be any history that ensures $L_{\Diamond\mathrm{p}}$ and T_k be any correct transaction in H. Let t be any time at which T_k is pending in H, and $C = Correct_H(Concurr_H(t))$. Clearly, T_k must be in set C. Because $L_{\Diamond\mathrm{p}}(C) = \{C\}$, $Committed_H(C)$ must be the entire set C, and so $T_k \in C$ must be committed in H.

($\Leftarrow$) Let H be any history in which every correct transaction is committed. Let t be any time and $C = Correct_H(Concurr_H(t))$. Because every transaction in C is committed, $Committed_H(C) = C \in L_{\Diamond\mathrm{p}}(C)$. □

Global progress. Intuitively, a TM implementation M ensures *global progress* (analogous to *lock-freedom* for shared-memory objects, when considered in a crash-prone system), if in every infinite history of M, in which there is a pending correct transaction, there are infinitely many committed transactions. More formally, global progress is the function $L_{\Diamond g}(C) = \{\{T_{i_1}\}, \{T_{i_2}\}, \dots\}$, where T_{i_1}, $T_{i_2}, \dots$ are all elements of set C.

In a crash-prone system, global progress is ensured by so-called *lock-free* TM implementations such as OSTM [7]. We give a simple TM implementation that guarantees global progress in Appendix A. In a crash-free system, global-progress could be ensured by a lock-based TM implementation; however, we do not know of any such TM (implementations such as TL2 [5], TinySTM [6], or SwissTM [2] allow livelock situations—scenarios in which two concurrent correct transactions are pending forever).

Theorem 6 *An infinite history H ensures $L_{\Diamond g}$ if, and only if, whenever there is a pending correct transaction in H, then infinitely many transactions are committed in H.*

Proof. ($\Rightarrow$) Let H be any infinite history that ensures $L_{\Diamond g}$. Assume that there is a pending correct transaction T_k in H. By contradiction, assume that after some time t no transaction commits. Because T_k is pending, set $Q = Correct_H(Concurr_H(t'))$, where $t' > t$, contains at least transaction T_k. Hence, by $L_{\Diamond g}$, some transaction from Q must be committed in H—a contradiction.

($\Leftarrow$) Let H be any infinite history. If H has no pending correct transaction, then H trivially ensures $L_{\Diamond g}$. Assume then that H contains a pending correct transaction T_k and there are infinitely many committed transactions in H. But then, for every time t, set $Q = Correct_H(Concurr_H(t))$ contains either (1) only committed transactions, or (2) some pending transactions and infinitely many committed transactions. In both cases property $L_{\Diamond g}$ is ensured. □

Solo progress. Intuitively, a TM implementation M ensures *solo progress* (analogous to *obstruction-freedom* for shared-memory objects, when considered in a crash-prone system), if in every infinite history H of M every correct transaction that eventually runs *alone* for sufficiently long time commits. The classical meaning of the term "alone" (as used by obstruction-freedom [3]) is "with no other transaction taking steps concurrently". Zombie transactions, however, have never been considered before in this context. In the following definition, we assume that a transaction T_i is alone if T_i is concurrent only to incorrect (crashed or zombie) transactions. In a system without zombie transactions, this is equivalent to saying "with no transaction other than T_i taking steps concurrently" (as we prove below). More formally, solo progress is the following function:

$$L_{\Diamond s}(C) = \begin{cases} \{C\} & \text{if } |C| = 1 \\ \{\emptyset\} & \text{otherwise} \end{cases}$$

In a crash-prone system without zombie transactions, solo progress is ensured by TM implementations such as DSTM [13], RSTM [16] (with its nonblocking backend), or NZTM [21]. In a crash-free system, solo progress is ensured by most (if not all) lock-based TM implementations, e.g., TL2 [5], TinySTM [6], or SwissTM [2]. (In fact, the progress semantics of those TMs, as formalized in [10], is stronger than solo progress in a crash-free system.) However, only lazy-acquire TMs, such as TL2, ensure solo progress with zombie transactions.

Theorem 7 *An infinite history H without zombie transactions ensures property $L_{\Diamond s}$ if, and only if, every correct transaction in H that from some point in time runs alone, i.e., without other transactions concurrently executing steps, eventually commits.*

Proof. ($\Rightarrow$) Let H be any infinite history without zombie transactions that ensures property $L_{\Diamond s}$. By contradiction, assume that there is a correct transaction $T_k \in H$, such that T_k executes steps alone from some time t but T_k is pending. That is, no transaction other than T_k executes any step after t. But then, no transaction that is concurrent to T_k is correct, and so $Correct_H(Concurr_H(t)) = \{T_k\}$. Hence, because $L_{\Diamond s}(\{T_k\}) = \{\{T_k\}\}$, T_k must be committed in H—a contradiction.

($\Leftarrow$) Let H be any infinite history without zombie transactions. If every correct transaction is committed in H, then H trivially ensures $L_{\Diamond s}$. Assume then that there is a pending correct transaction T_k in H and that, for every time t, some transaction other than T_k takes a step after t. But then, for every time t at which T_k is pending, set $Correct_H(Concurr_H(t))$ contains some transaction other than T_k, and so $L_{\Diamond s}$ is ensured. □

3.3 Classes of TM Liveness Properties

Intuitively, we say that a TM liveness property L is *nonblocking* if L ensures progress for every transaction that runs alone, i.e., with no concurrent correct transactions. More formally, we say that a TM liveness property L is nonblocking if, for every transaction $T_i \in \mathcal{T}$, $L(\{T_i\}) = \{\{T_i\}\}$.

Intuitively, we say that a TM liveness property L is *well-formed* if L does not guarantee more when the number of concurrent transactions increases. More precisely, we say that L is well-formed if, for every finite set $Q \subset \mathcal{T}$, every subset Q' of Q, and every element S in $L(Q)$, set $S \cap Q'$ is a subset of some element in $L(Q')$.

Local progress, global progress, and solo progress are all nonblocking and well-formed TM liveness properties.

4 Ensuring TM Liveness with Crashes

In this section, we prove that the strongest nonblocking property that can be ensured in a crash-prone system is global progress. We first identify a class of TM liveness properties that we call $(n-1)$*-prioritizing*. Intuitively, every $(n-1)$-prioritizing TM liveness property L is characterized by an infinite set $C \subseteq \mathcal{T}$ and a subset

$P \subset C$ of size $n-1$. Then, if transactions in set C are correct and (indirectly) concurrent in some history H (i.e., $\mathit{Correct}_H(\mathit{Concurr}_H(t)) = C$ at some time t), then for history H to ensure L at least one of the transactions in set P must be committed in H. In a sense, P is a set of transactions with higher priority, and one of those transactions has to commit in the (single) scenario described by set C.

We then prove that (1) every TM liveness property that is *not* $(n-1)$-prioritizing is weaker than global progress, and (2) a nonblocking TM liveness property L can be ensured in a crash-prone system if, and only if, L is *not* $(n-1)$-prioritizing.

More formally, let L be any TM liveness property. We say that L is *$(n-1)$-prioritizing*, if there exists an infinite subset C of set $\mathcal{T}$ and a subset P of C of size $n-1$, such that, for every non-empty set S in $L(C)$, $P \cap S \neq \emptyset$.

Theorem 8 *Every TM liveness property that is not $(n-1)$-prioritizing is weaker than $L_{\Diamond g}$.*

Proof. Let L be any nonblocking TM liveness property. By contradiction, assume that L is not $(n-1)$-prioritizing and L is not weaker than $L_{\Diamond g}$.

Because L is not weaker than $L_{\Diamond g}$, there exists a history H such that H ensures $L_{\Diamond g}$ and H does not ensure L. Because H does not ensure L, there is a time t such that, if $C = \mathit{Correct}_H(\mathit{Concurr}_H(t))$, then $\mathit{Committed}_H(C) \not\supseteq S$ for every $S \in L(C)$. Note first that if C is a finite set, then $L_{\Diamond g}$ requires that all transactions in C are committed, i.e., that $\mathit{Committed}_H(C) = C$. That is, if C is a finite set, then L cannot be violated in H at time t. Hence, C is an infinite set. Denote by P the set of transactions in C that are pending in H. Because H ensures $L_{\Diamond g}$ and because at most n transactions can be concurrent, the size of set P is at most $n-1$. Clearly, all transactions in set $C-P$ are committed in H.

Let P' be any set such that $P \subseteq P' \subseteq C$ and the size of P' is $n-1$. Because L is not $(n-1)$-prioritizing, there exists an element $S \in L(C)$ such that $P' \cap S = \emptyset$. But then, because $S \subseteq C$, set $\mathit{Committed}_H(C) = C - P \supseteq C - P'$ is a superset of S—a contradiction. ☐

Before we prove the other key theorem of this section, we prove the following two auxiliary lemmas (each, in fact, interesting in its own right).

Lemma 9 *For every TM implementation M that ensures any nonblocking TM liveness property, and for every pair of sets P and C, where $P \subset C \subseteq \mathcal{T}$, $|C| = \infty$, and $|P| = n-1$, there exists an infinite history H of M and a time t such that $\mathit{Correct}_H(\mathit{Concurr}_H(t)) = C$ and all transactions from set P are correct and pending in H.*

Proof. Let M be any TM implementation that ensures some nonblocking TM liveness property L. Recall that n, the number of processes, is also the maximum number of transactions that can be concurrent at any time. Consider a history H of M generated in the following execution (initially, $k = 0$; x is some t-object initialized to 0):

1. Transactions $T_1, \ldots, T_{n-1}$ read x.
2. Transaction T_{n+k} reads some value v from t-object x and writes value $1-v$ to x. Then, T_{n+k} attempts to commit. If T_{n+k} aborts, T_{n+k} is restarted and executed until it commits. No transaction executes steps concurrently to T_{n+k}.
3. Transactions $T_1, \ldots, T_{n-1}$ write value $1-v$ to x and attempt to commit. If all of them abort, go to step 1 with $k \leftarrow k+1$.

Assume first, by contradiction, that, for some k, transaction T_{n+k} is pending in H. But then, transactions T_1, $\ldots$, T_{n-1} are faulty in H, and so $\mathit{Correct}_H(\mathit{Concurr}_H(t)) = \{T_{n+k}\}$ for some time t. Hence, as M ensures L that is nonblocking, T_{n+k} must be committed in H—a contradiction.

Assume then, by contradiction, that some transaction T_m, $1 \leq m \leq n-1$, commits. Let T_{n+w} be the latest transaction T_{n+k} that precedes T_m, and v_w be the value written to t-object x by T_{n+w}. Until T_m reads x, there is no concurrent transaction that writes to x. Hence, because M ensures opacity and because the future operations of transactions are not known in advance to the TM, T_m must read v_w from x. Then, T_m writes value $1-v_w$ to x and commits. But transaction T_{n+w+1} also reads value v_w from x and writes $1-v_w$ to x, and no transaction concurrent to T_m or T_{n+w+1} writes back value v_w to x. Hence, there is no way to order transactions T_m and T_{n+w+1}, and so opacity is violated—a contradiction.

Therefore, history H of M is infinite, and H contains $n-1$ correct and pending transactions $T_1, \ldots, T_{n-1}$. ☐

Lemma 10 *There exists a TM implementation that ensures global progress in a crash-prone system with zombie transactions.*

OSTM [7] is a TM implementation that ensures global progress in a crash-prone system, even with zombie transactions. However, we do not know whether this has been formally proved. Therefore, for completeness, we present in Appendix A a simple TM implementation, and we prove that this TM ensures opacity and global progress in a crash-prone system with zombie transactions. (It is worth noting that proving this is a technical challenge even for such a simple algorithm.)

Theorem 11 *A nonblocking TM liveness property L can be ensured by a TM implementation in a crash-prone system if, and only if, L is not $(n-1)$-prioritizing.*

Proof. ($\Rightarrow$) Let L be any nonblocking, $(n-1)$-prioritizing TM liveness property. Hence, there exists an infinite set $C \subseteq \mathcal{T}$ and a set $P \subset C$ of size $n-1$ such that $P \cap S \neq \emptyset$ for every $S \in L(C)$. By contradiction, assume that there is a TM implementation M that ensures L in a crash-prone system. By Lemma 9, and because L is nonblocking, there exists a history H of M and a time t such that $\mathit{Correct}_H(\mathit{Concurr}_H(t)) = C$ and all transactions from set P are correct and pending in H. But then, for every

$S \in L(C)$, $P \cap S \neq \emptyset$ and so $Committed_H(C) \not\supseteq S$. Hence, H violates L—a contradiction.

($\Leftarrow$) Let L be any nonblocking TM liveness property that is not $(n-1)$-prioritizing. By Theorem 8, L is weaker than $L_{\Diamond g}$. But, by Lemma 10, $L_{\Diamond g}$ can be implemented in a crash-prone system. Hence, L can also be ensured in a crash-prone system. □

Corollary 12 *Global progress is the strongest nonblocking TM liveness property that can be ensured by any TM in a crash-prone system.*

5 Zombie Transactions

In this section, we prove that, in the computability sense, zombie transactions are neither easier nor more difficult to deal with than crashed transactions. We first show how to transform a TM implementation that works only in a crash-free system but tolerates zombie transactions into a one that works in a crash-prone system (and ensures the same TM liveness property). The transformation assumes well-formed TM liveness properties (we discuss this assumption in Section 6). Given that, as we showed in Section 4, global progress can be implemented in a crash-prone system with zombie transactions, the strongest well-formed, nonblocking TM liveness property that can be ensured in a crash-prone system (with or without zombie transactions) is also global progress.

Theorem 13 *Given any TM implementation M that ensures any well-formed TM liveness property L in a crash-free system with zombie transactions, we can devise a TM implementation M′ that ensures L in a crash-prone system (with zombie transactions).*

Proof. (sketch; a full proof is in Appendix B) Let L be any TM liveness property and M be any TM implementation that ensures L in a crash-free system with zombie transactions. For simplicity, but without loss in generality, assume that M is a deterministic algorithm. We build a TM implementation M' (Algorithms 1 and 2) that ensures L in a crash-prone system (with zombie transactions).

The intuition behind implementation M' is the following. We replicate M into every process, i.e., we make each process p_i execute its local copy M_i of M. Within the instance M_i, process p_i simulates every process p_m (coroutine *simulate*(m)) by executing steps of the simulated processes in a round-robin, deterministic way (line 4). (For simplicity, we assume within this proof that events are also steps.)

The execution of M' is divided into rounds. In each round, processes simulate a single process p_m (common to all processes, because M' is deterministic) executing a single operation in an instance of TM implementation M. The processes first agree (using consensus[5] objects

[5]A consensus object implements an operation *propose* that takes a *proposed* value as an input, and returns a *decision* value. It allows processes to agree (decide) on a single value chosen from the values those processes have proposed.

Algorithm 1: A transformation of a TM implementation M that tolerates zombie transactions into a TM implementation M' that tolerates crashed and zombie transactions (code for process p_i); continues in Algorithm 2.

```
uses: $C[1,\ldots,n][1,\ldots]$—(infinite) array of consensus
        objects, $T[1,\ldots,n]$—array of registers,
        $M_i$—instance of $M$ local to $p_i$ (other variables
        are also local to $p_i$)
initially: $T[1,\ldots,n] = (\bot, 0)$, $last\text{-}ts = 0$, $round = 1$,
        $ts = 1$

1  upon operation $op$ by transaction $T_k$ do
2      $T[i] \leftarrow (op, ts)$; $ts \leftarrow ts + 1$;
3      $retval \leftarrow \bot$;
4      while $retval = \bot$ do  for $m \leftarrow 1$ to $n$ do execute
       next step of coroutine simulate($m$);
5      return $retval$;

6  coroutine simulate($m$)
7      $op \leftarrow$ get-next-operation($m$, false);
8      $s \leftarrow$ simulate-operation($m$, $op$);
9      if $s$ is a commit event then goto line 7;
10     if $s$ is an abort event then goto line 23;

11     $op \leftarrow$ get-next-operation($m$, true);
12     $s \leftarrow$ simulate-operation($m$, $op$);
13     if $s$ is a commit event then goto line 7;
14     if $s$ is an abort event then
15         if $op \neq \bot$ then goto line 11;
16         while true do
17             $op \leftarrow$ get-next-operation($m$, true);
18             if $op \neq \bot$ then break;
19             simulate-operation($m$, $op$);

20         if $m = i$ then $retval \leftarrow A_k$;
21         goto line 23;
22     goto line 11;

23     $op \leftarrow$ get-next-operation($m$, true);
24     if $op \neq \bot$ then
25         simulate-operation($m$, $\langle tryA \rangle$);
26         goto line 12;
27     $s \leftarrow$ simulate-operation($m$, $\bot$);
28     goto line 23;
```

in array C) on the next operation to be executed by p_m, i.e., an operation of a transaction at p_m (function *get-next-operation*), and then they execute this operation (function *simulate-operation*). The steps executed by a process p_i in a given round, on behalf of a simulated process p_m, are interleaved (in a deterministic way; see line 4) with the steps executed by p_i in other rounds on behalf of simulated processes other than p_m. M_i. Note also that different processes may be in different rounds at any given time.

When any transaction T_k, executed by a process p_i, invokes an operation op on M', process p_i first announces

Algorithm 2: The second part of Algorithm 1

```
function get-next-operation($m$, allow-dummy-op)
    repeat
        simulate one step of process $p_m$ in $M_i$;
        $(op, ts) \leftarrow T[m]$;
        if $ts \leq$ last-ts then $(op, ts) \leftarrow (\bot, \bot)$;
        $(op, ts) \leftarrow C[m, round].propose((op, ts))$;
        round $\leftarrow$ round $+ 1$;
    until allow-dummy-op or $op \neq \bot$ ;
    if $op \neq \bot$ then last-ts $\leftarrow ts$;
    return $op$;

function simulate-operation($m$, $op$)
    if $op \neq \bot$ then simulate invocation of $op$ by process $p_m$ in $M_i$;
    else simulate invocation of dummy operation $\langle read\ x \rangle$ by process $p_m$ in $M_i$;
    repeat
        simulate one step $s$ of process $p_m$ in $M_i$;
    until $s$ is a response event ;
    if $m = i$ and $op \neq \bot$ then
        retval $\leftarrow$ return value in $s$;
    return $s$;
```

this operation (line 2). Then, p_i continues (or starts) simulating all processes within its instance M_i of M (line 4). Operation *op* is eventually decided in some round r by consensus object $C[r]$ (line 6), and executed within M_i. Once *op* returns in M_i, the return value is stored in variable *retval* and eventually returned to transaction T_k (line 5).

Because TM implementation M does not tolerate crashes, we need to ensure, at every correct process p_i that executes infinitely many steps of M, that (1) every process simulated by p_i in instance M_i executes infinitely many steps, and (2) every transaction that is pending forever, and that is not blocked by M_i inside an operation infinitely long, executes infinitely many operations. Property (1) is satisfied because p_i simulates steps of all processes within M_i in a round-robin fashion (and because p_i is correct).

To ensure property (2), we make p_i periodically insert a *dummy* operation (*read x*, for some t-object x) into every pending transaction that does not announce a new operation in array T (line 27). A dummy operation does not change the state of any t-object. If a dummy operation inserted into a transaction T_k returns an abort event, the abort event is propagated to T_k as a response to the next operation invoked by T_k (line 20).

There is one subtlety here. If a dummy operation is inserted into a transaction T_k at the beginning of some sub-transaction T_k^m of T_k, then a sub-transaction T_j^l that precedes T_k^m in a history of M' can become concurrent to T_k^m in some instance of M. In this case, M' could violate opacity. We prevent this problem in the following way. If one or more dummy operations are executed by T_k (in some instance M_i of M) just after an operation of T_k that returns an abort event (i.e., at the beginning of a sub-transaction of T_k), and T_k issues an operation *op* on M', we first make T_k execute (in M_i) operation $tryA(T_k)$. Hence, we create a separate sub-transaction of T_k, which we call a *dummy sub-transaction*, that consists only of dummy operations followed by operation *tryA*. Once a dummy sub-transaction is aborted, operation *op* of T_k can be executed in M_i, within a new sub-transaction. □

Corollary 14 *The strongest nonblocking, well-formed TM liveness property that can be ensured by any TM implementation in a system with zombie transactions is global progress.*

6 Discussion

This paper addresses the question of how much liveness a TM implementation can ensure. We define precisely the notion of a TM liveness property, prove that the strongest (nonblocking) TM liveness property that can be ensured in an asynchronous system with transaction crashes is global progress, and show that considering zombie transactions does not change in a fundamental way the implementability of TM liveness properties. As we pointed out in the introduction, these are preliminary steps towards understanding TM liveness. In the following, we discuss the major assumptions underlying our results, and present some open questions.

First, when proving that every TM liveness property L that is $(n-1)$-prioritizing is impossible to implement in a crash-prone system, we assumed that L is nonblocking. There are indeed TM liveness properties that are $(n-1)$-prioritizing, blocking, and impossible to implement with transaction crashes, e.g., a property L such that: $L(C) = \{\emptyset\}$ if C contains a transaction T_1 and $L(C) = \{C\}$ otherwise. However, there are also TM liveness properties that are $(n-1)$-prioritizing (and blocking) and that can be implemented in a crash-prone system, e.g., $L(C) = \{C\}$ if $C = Q$ and $L(C) = \{\emptyset\}$ otherwise, where Q is some predefined infinite set of transactions: all transactions in Q have high-priority—they must all be committed in the execution defined by set Q, but a TM implementation may simply prevent such an execution by blocking or aborting some transactions forever. In particular, a TM that blocks every transaction infinitely long ensures L, yet it can trivially be implemented in a crash-prone system.

Second, the transformation from a TM implementation M that ensures some property L in a crash-free system with zombie transactions to a TM implementation M' that ensures L in a crash-prone system assumed that L is well-formed. The transformation does not work for instance if L ensures progress for a transaction T_k only if T_k is concurrent to some other transaction, and not when T_k runs alone. This is because the transactions that are concurrent in the transformed implementation M' might be executed sequentially by the base implementation M. On the one hand, it is difficult to see any useful TM liveness property that would not be well-formed and nonblocking. Indeed, the TM liveness properties that we present in this paper, and which are ensured by most TM implementations to date, are all well-formed and nonblocking. On the other hand, it is intriguing to determine the precise impact of those assumptions on the results presented in this paper.

Another interesting direction is related to complexity. Indeed, we proved the equivalence of crashed and zombie transactions. However, the inherent cost, in terms of time and space complexity, of ensuring a given TM liveness property might be different for systems with crashed and zombie transactions. Whether it is indeed the case is an open question.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[2] A. D. Rachid Guerraoui and M. Kapałka. Stretching transactional memory. In *PLDI*, 2009.

[3] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC*, 2005.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[6] P. Felber, T. Riegel, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[7] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2003.

[8] R. Guerraoui and M. Kapałka. On obstruction-free transactions. In *SPAA*, 2008.

[9] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *PPoPP*, 2008.

[10] R. Guerraoui and M. Kapałka. The semantics of progress in lock-based transactional memory. In *POPL*, 2009.

[11] R. Guerraoui, M. Kapałka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.

[12] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[15] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, 2005.

[16] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT*, 2006.

[17] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java stm. In *SPAA*, 2008.

[18] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, 2008.

[19] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[20] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[21] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: nonblocking zero-indirection transactional memory. In *TRANSACT*, 2007.

[22] J. Vitek, S. Jagannathan, A. Welc, and A. Hosking. A semantic framework for designer transactions. In *ESOP*, 2004.

Appendix

A A TM Implementation that Ensures Global Progress in a Crash-Prone System with Zombie Transactions

An example TM implementation that ensures global progress in a crash-prone system (with zombie transactions) is shown in Algorithm 3. (Note that the algorithm is not meant to be practical—it is presented here for the sole purpose of proving Lemma 10.) The intuition behind the algorithm is the following (a proof of correctness follows). When a sub-transaction T_k^l executed by a process p_i invokes its first operation, p_i takes a snapshot of all current states of t-objects and stores those states in the next available slot of array S (lines 7–8). Process p_i searches for an available slot s by scanning array A of test-and-set objects[6] (lines 4–6). If $A[s] = 1$, then slot s is being used by some process, and exclusive to this process; otherwise it is available. Once p_i verifies that the snapshot is consistent (line 9), p_i can execute all subsequent operations of T_k^l on the snapshot.

If T_k^l invokes operation $tryC(T_k)$, then p_i tries to atomically change the current snapshot by updating the value (state) of compare-and-swap object[7] C to point to the slot of p_i in array S (line 14). The update will be successful only if no other process committed a transaction concurrently to T_k^l. If p_i succeeds in updating C, p_i releases the slot of S that contains the old snapshot of t-object states (the one p_i read at the beginning of sub-transaction T_k^l). Otherwise, p_i releases its own slot.

Denote Algorithm 3 by M. We prove that M ensures opacity and global progress in a crash-prone system with zombie transactions.

Opacity. Let H be any history of M. Observe first that each object $A[s]$ acts as a lock for the registers $S[s][1,\ldots,K]$. That is, if a sub-transaction T_k^l is returned 1 from operation *test-and-set* invoked in line 4 on object $A[s]$, then no other sub-transaction can modify any register $S[s][1,\ldots,K]$ until T_k^l executes line 16 or line 20. Hence, registers $S[s][1,\ldots,K]$ can be thought of as local to T_k^l during all operations of T_k^l that return values different than A_k and C_k.

Therefore, we can view any sub-transaction T_k^l (executed by a process p_i) in H as a sequence of *read* operations on all t-objects (reading $S[c_k^l][1,\ldots,K]$ in line 8), and a sequence of one or more *write* operations on every t-object x_m (writing $S[slot_i][m]$ in line 8 and line 10). Hence, T_l^k first reads from every t-object x_m, then writes to every t-object x_m, and then writes to some t-objects. Without loss in generality, we can assume that each value written to a t-object is unique, i.e., that we can identify the writer transaction of every value read by a transaction. We prove that H ensures opacity by using the graph characterization of opacity introduced in [9].

Let Q denote the set of sub-transactions in H that received value *true* from the *compare-and-swap* operation executed in line 14. (Clearly, every sub-transaction that is committed in H is in Q.) Let Q' denote the set of non-committed sub-transactions in Q.

Let T_k^l be any sub-transaction executed by any process p_i. Denote by c_k^l and v_k^l the values of variables $curr_i$ and ver_i read by p_i in line 7 within the first operation of T_k^l (assume $v_k^l = \infty$ if T_k^l has not executed line 7

Algorithm 3: A TM implementation that ensures $L_{\Diamond g}$ (code for each process p_i; $x_1,\ldots,x_K$ are t-objects implemented by the algorithm)

```
uses: $A[1,\ldots,n+1]$—array of test-and-set objects,
      $S[1,\ldots,n+1][1,\ldots,K]$—array of registers,
      $C$—unbounded compare-and-swap object
      (other variables are local to process $p_i$)
initially: $A[1] = 1$, $A[2,\ldots,n+1] = 0$, $S[1][m] =$ the
           initial state of t-object $x_m$ (for $m = 1,\ldots,K$),
           $C = (1,1)$, $slot_i = \bot$ (at every process $p_i$)

1  upon operation op on t-object $x_m$ by transaction $T_k$ do
2      if $slot_i = \bot$ then
3          $slot_i = 1$;
4          while $A[slot_i].test\text{-}and\text{-}set = 1$ do
5              $slot_i \leftarrow slot_i + 1$;
6              if $slot_i > n+1$ then return $abort(T_k)$;
7          $(curr_i, ver_i) \leftarrow C.read()$;
8          for $r = 1$ to $K$ do $S[slot_i][r] \leftarrow S[curr_i][r]$;
9          if $C.read() \neq (curr_i, ver_i)$ then return
           $abort(T_k)$;
10     return $S[slot_i][m].op$;

11 upon tryA do
12     return $abort(T_k)$;

13 upon tryC do
14     $s \leftarrow C.compare\text{-}and\text{-}swap((curr_i, ver_i),$
       $(slot_i, ver_i + 1))$;
15     if not $s$ then return $abort(T_k)$;
16     $A[curr_i].reset()$;
17     $slot_i \leftarrow \bot$;
18     return $C_k$;

19 function $abort(T_k)$
20     if $slot_i \neq \bot$ and $slot_i \leq n+1$ then $A[slot_i].reset()$;
21     $slot_i \leftarrow \bot$;
22     return $A_k$;
```

[6] A test-and-set object implements operations: (1) *test-and-set* that atomically reads the state of the object, changes the state to value 1, and returns the state read, and (2) *reset* that sets the state of the object to 0.

[7] A compare-and-swap object implements an operation *compare-and-swap*(v, v') that atomically changes the state of the object from value v to v'; the operation returns *true* if the change was successful, and *false* otherwise. It is also possible to read the state of a compare-and-swap object.

within its first operation in H). Let $\ll$ be any total order on sub-transactions in H such that, for every two sub-transactions T_k^m and T_j^l in H, if (1) $T_k^m \in Q$ and $v_k^m < v_j^l$, (2) $T_j^l \in Q$ and $v_k^m \leq v_j^l$, or (3) T_k^m precedes T_j^l in H, then $T_k^m \ll T_j^l$. It is straightforward to see that such a total order exists. Indeed, (1) if T_k^m precedes T_j^l in H then $v_k^m \leq v_j^l$, if $T_k^m \notin Q$, or $v_k^m < v_j^l$ if $T_k^m \in Q$, and (2) if $v_k^m = v_j^l$, then T_k^m and T_j^l cannot be both in Q (i.e., they cannot be both returned *true* in line 14).

Let G be the opacity graph $OPG(H, \ll, Q')$. History H ensures opacity if, and only if, G is well-formed and acyclic. (For the definitions of the terms we use here, refer to [9].)

Claim 15 *If the state of object C is $(c, v) \neq (1, 1)$ at some time t, then every value in $S[c][1, \ldots, K]$ at time t has been written by a sub-transaction that was returned value true in line 14 before t.*

Proof. The claim trivially holds while $C = (1,1)$, i.e., C is in its initial state. Assume that the state of C at some time t is (c, v), and that every value in $S[c][1, \ldots, K]$ at time t is indeed a value written by some sub-transaction T_k^l that was returned value *true* in line 14 before t. Let t' be at time at which the state of C is changed by some sub-transaction T_w^u from (c, v) to (c', v'). Because registers $S[c'][1, \ldots, K]$ are all written to by T_w^u and cannot be changed by any other sub-transaction until time t', and because T_w^u must be returned value *true* in line 14 before t', the claim also holds at t'.

Let then t'' be any time between t and t'. Sub-transaction T_k^l must have set the state of $A[c]$ to 1, and T_k^l could not change $A[c]$ thereafter. The state of $A[c]$ can be changed only by a sub-transaction that changes the state of C. Hence, $A[c] = 1$ at t''. But then no sub-transaction can have its *slot* variable equal to c at t'', and so no sub-transaction can change any value in $S[c][1, \ldots, K]$ at time t''. Hence, the claim holds also at t'' and, by extension, at any time. $\square$

By contradiction, assume that G is not well-formed. That is, there is a sub-transaction T_k^l that reads some value q from some register $S[c_k^l][m]$, and q is written to $S[c_k^l][m]$ by some sub-transaction T_w^u that is not in set Q. But then, because T_k^l reads c_k^l in line 7, and by Claim 15, T_w^u must be in set Q—a contradiction.

By contradiction, assume that there is a cycle L in G. Hence, there are some two sub-transactions T_k^l and T_w^u such that $T_k^l \ll T_w^u$ and there is an edge from T_w^u to T_k^l. Clearly, the edge cannot be labelled L_{rt} because if T_w^u precedes T_k^l in H, then $T_w^u \ll T_k^l$.

Assume that T_k^l reads from some register $S[c_k^l][m]$ value q that is written by T_w^u. Hence, T_w^u must be in set Q. Clearly, it is impossible that T_k^l precedes T_w^u, as then T_k^l would read q before T_w^u event starts. But then, by Claim 15 and because T_w^u increases the version number field of C when T_w^u executes line 14, $v_w^u < v_k^l$—a contradiction with the assumption that $T_k^l \ll T_w^u$.

Assume then that there is an edge labelled L_{ww} from T_w^u to T_k^l. That is, T_w^u is in set Q, and there is a sub-transaction T_z^x in H such that $T_w^u \ll T_z^x$, and T_z^x reads from some register $S[c_z^x][m]$ a value q that is written to $S[c_z^x][m]$ by T_k^l. Observe first that if $v_z^x = v_w^u$, then T_z^x cannot be in set Q. Hence, because $T_w^u \ll T_z^x$, $v_w^u < v_z^x$. Because T_z^x reads value q that is written by T_k^l, sub-transaction T_k^l must be in set Q and T_k^l must execute line 14 after T_w^u executes line 14. But then, v_k^l must be larger than v_w^u—a contradiction with the assumption that $T_k^l \ll T_w^u$.

Global progress. By contradiction, assume that there is a history H of M that violates global progress. That is, there is a time t, such that every transaction from set $C = Correct_H(Concurr_H(t))$ is pending in H (and $C \neq \emptyset$). Hence, no transaction commits after t.

Let T_k be any transaction in C, executed by some process p_i, and T_k^m be any sub-transaction of T_k that invokes its first operation after t. Observe first that T_k^m cannot be blocked by M inside any operation infinitely long. Hence, because T_k is a correct transaction, T_k^m must be aborted.

Let c_k^m be the value read by p_i executing T_k^m from object C in line 7. Because no transaction commits after time t, no process changes the state of object C after t. Hence, when T_k^m reaches line 14, C still contains value c_k^m. Therefore, T_k^m cannot abort in line 6 and T_k^m must be returned *true* from operation *compare-and-swap* in line 14, and so T_k^m cannot abort—a contradiction.

B Proof of Correctness of the Transformation Shown in Algorithms 1 and 2

We prove here the correctness of the TM implementation M' shown in Algorithms 1 and 2, which transforms a TM implementation M that ensures a well-formed TM liveness property L in a crash-free system with zombie transaction into a TM implementation M' that ensures L in a crash-prone system.

Opacity. Let H be any history of TM implementation M', and T_k be any transaction in H executed by some process p_i. Observe first that if T_k invokes an operation *op* and is returned a value v which is different than A_k (an abort event of T_k), then T_k also invokes *op* in instance M_i and is returned v from M_i. Moreover, if T_k invokes an operation *op'* after executing *op*, and *op'* returns a non-A_k value, then also in instance M_i transaction T_k executes first *op*, and then *op'*. It is possible that in M_i transaction T_k executes some dummy operations between *op* and *op'*; however, none of those dummy operations returns A_k—otherwise, *op'* would also be returned A_k.

Therefore, for every sub-transaction T_k^m of T_k there is a *corresponding* sub-transaction T_k^c in history H_i of instance M_i, i.e., a sub-transaction T_k^c such that T_k^c invokes its first operation after T_k^m invokes its first operation, T_k^c returns from its last operation before T_k^m returns from its last operation, and T_k^c executes all the operations of T_k^m (in the same relative order and returning the same values) that did not return value A_k. Moreover, T_k^c is aborted if, and only if, T_k^m is aborted or pending (if T_k^c is returned A_k from a dummy operation, then T_k^m is returned A_k from its next operation).

Let *op* and *op*′ be any two executions of some operations (possibly by different transactions) in history H. Assume that *op*′ is invoked after *op* returns, and that both *op* and *op*′ return events assigned to variable *retval* in line 17. Hence, both *op* and *op*′ have to be decided by some consensus objects $C[r]$ and $C[r']$, respectively. But because processes traverse array C always towards larger round numbers, and because every consensus object can decide only one operation, r must be lower than r'. Therefore, the order of those operations in H that do not return an abort event assigned in line 20 is preserved in the history of every instance M_i, $i = 1, \ldots, n$.

The algorithm of M' is deterministic, each instance M_i, $i = 1, \ldots, n$, is deterministic, and processes agree on the order of invocation events through the consensus array C. Therefore, for any two instances M_i and M_j, $i, j = 1, \ldots, n$, either the history of M_i is a prefix of the history of M_j or vice versa. Let then M_s be an instance which history H_s has the maximum length. Because M_s ensures opacity, H_s also ensures opacity.

Let T_j^l and T_k^m be any sub-transactions in H such that T_j^l precedes T_k^m. Let T_j^u and T_k^w be the sub-transactions in H_s corresponding to, respectively, T_j^l and T_k^m. If T_j^l is committed or T_j^l is returned an abort event assigned to variable *retval* in line 17, then it is easy to see that T_j^u must precede T_k^w in H_s. Indeed, the last operation of T_k^m and the first operation of T_j^l must return an event returned by M_s (and assigned to *retval* in line 17), and so the order of those operations must be the same in H_s as in H. Assume then that T_j^l is returned an abort event assigned to *retval* in line 20. (Note that T_j^l cannot be pending in H, as then T_j^l could not precede T_k^m.) Hence, the last operation of sub-transaction T_j^u is a dummy operation that returns an abort event. This dummy operation must be decided by some consensus object $C[r]$, and must be executed by process $p(T_j^l)$ before an abort event is returned to T_j^l. Because the first operation *op* of T_k^m is invoked after the last event of T_j^l, and because *op* cannot return an abort event assigned to *retval* in line 20, operation *op* has to be decided by some consensus object $C[r']$ such that $r' > r$. Hence, sub-transaction T_j^u must precede T_k^w in H_s.

Let H' be a history obtained from H by replacing every invocation event of an operation that returns an abort event assigned to variable *retval* in line 20 with an invocation event of a dummy operation. Clearly, H ensures opacity if, and only if, H' ensures opacity. Indeed, if an operation returns an abort event, its semantics is the same regardless of the operation. Let H'_s be a history obtained from H_s by removing events of (1) all dummy operations that do not return an abort event and (2) all dummy transactions. A dummy operation cannot be the first operation of a non-dummy sub-transaction in H_s, and a dummy operation that does not return an abort event cannot be the last operation of any committed or aborted sub-transaction in H_s. Hence, the precedence relation among non-dummy transactions in H_s is the same in H'_s. Therefore, because dummy operations and dummy transaction do not change the state of any t-object, and because history H_s ensures opacity, H'_s must also ensure opacity. But then, history H must ensure opacity, because $H'|p_i = H'_s|p_i$ for every process p_i (i.e., histories H' and H'_s are equivalent), and the precedence relation between transactions in H' is preserved in H'_s.

TM liveness. By contradiction, assume that M' does not ensure TM liveness property L. That is, there is a history H of M' and a time t, such that, if $Q = \mathit{Correct}_H(\mathit{Concurr}_H(t))$ and $C = \mathit{Committed}_H(Q)$, then C is not a superset of any element of $L(Q)$. Let P be the set of pending transactions in Q. Clearly, P contains at least one transaction; otherwise, L could not have been violated in H.

Consider any transaction $T_k \in Q$ executed by some process p_i. When T_k invokes an operation *op*, operation *op* is announced in register $T[i]$ with some timestamp *ts* (larger than the timestamp of a previously announced operation at p_i, if any) at some time t. Hence, every process p_m that invokes function *get-next-operation* within coroutine $\mathit{simulate}(i)$ after time t will read in line 4 value $(\mathit{op}, \mathit{ts})$ and will not reject the value in line 5. Therefore, eventually value $(\mathit{op}, \mathit{ts})$ must be decided by some consensus object $C[r]$ (by the properties of consensus), and so process p_i eventually executes *op* on behalf of T_k in instance M_i.

Consider any process p_m that executes any of the transactions in set P. Clearly, p_m goes through infinitely many rounds, and eventually is returned in line 6 value $(\mathit{op}, \mathit{ts})$ from consensus object $C[r]$. Hence, p_m eventually invokes operation *op* on behalf of transaction T_k in instance M_m. That is, every transaction in Q is executed in instance M_m at every process p_m that executes a transaction from set P.

Let p_m be any process that executes any transaction T_k in set P. Let H_m denote the history of instance M_m at p_m. Note first that $H_m|p_s$ is infinite for every simulated process p_s and that every transaction that is pending in H_m either performs infinitely many operations or executes infinitely many steps within some operation that never returns. This is because process p_m is correct, and p_m executes steps on behalf of every (simulated) process in instance M_m. Moreover, if a transaction T_l does not announce any operation in array T after some time, then p_m assumes the dummy operation *read x* to be executed periodically by T_l. Hence, every transaction that crashes becomes a zombie in history H_m.

Because M_m eventually executes all transactions from set Q, and M_m eventually does not execute any transactions that are not from set Q, there exists a time t_m, such that set $Q_m = \mathit{Correct}_{H_m}(\mathit{Concurr}_{H_m}(t_m))$ is a subset of Q. Clearly, P is a subset of Q_m. Hence, because L is a well-formed TM liveness property, $\mathit{Committed}_{H_m}(Q_m)$ (which does not contain any transaction from P) cannot be a superset of any element of $L(Q_m)$. Therefore, M_m violates L—a contradiction.