How Live Can a Transactional Memory Be?*

Rachid Guerraoui   Michał Kapałka

February 2009 (updated: October 2009)

Abstract

Despite the large amount of work devoted so far to transactional memories (TMs), little is known about their *liveness*. Yet, the liveness of a system conveys its robustness and it is important to determine how live a TM can be. In this paper, we consider a system where transactions can be bogus or arbitrarily delayed and ask what can be ensured for correct ones. We determine a precise boundary between the liveness properties that can be ensured by a TM in such a system and those that cannot. In particular, we show that ensuring *local progress*—a TM liveness property analogous to wait-freedom or starvation-freedom in shared-memory implementations—is impossible. Indeed, we show that *global progress* is in a precise sense the strongest TM liveness property that a TM can guarantee. This property is analogous to lock-freedom for shared-memory objects and is indeed ensured by certain TM implementations.

1 Introduction

Transactional memory (TM) [15, 21] is a promising paradigm that aims at bringing concurrent programming to non-expert programmers. 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. It does also have some potential for exploiting underlying multi-core architectures as efficiently as hand-crafted, fine-grained locking, which is often an engineering challenge. Not surprisingly, a large body of work has been dedicated to implementing the TM paradigm and reducing its overheads. To a large extend, however, setting the theoretical foundations of the TM concept has been neglected.

Some recent work has been nevertheless devoted to formally defining the exact semantics of TM. More specifically, a correctness condition for TMs has been proposed in [11], and the programming language level semantics of specific classes of TM implementations has been determined, e.g., in [23, 16, 1, 19, 18]. A closer look at those papers, however, reveal that they all focus on *safety*, i.e., on what TMs *should not do*. Clearly, a TM that ensures only a safety property can trivially be implemented by blocking all operations or aborting all transactions. To be meaningful, a TM has to ensure some *liveness* [2], i.e., a guarantee about what *should be done*.

1.1 Liveness of a TM

In classical shared-memory systems, a liveness property describes when a process that invokes an operation on a shared object is guaranteed to return from this operation. For example, wait-freedom [13] ensures, intuitively, that *every* process invoking an operation will eventually return from this operation, even when other processes are suspended or crashed. In a TM, the question of liveness does not only apply to individual operations of transactions—indeed, a process which transactions are always aborted by a TM implementation, e.g., because of conflicts, does not really make any progress, even if it never blocks inside any operation. To make sense, a TM liveness property should ensure transaction commitment, not only operation termination.

Ideally, a TM would ensure that every process that keeps executing transactions eventually commits a transaction—a property that we call *local progress* and that is similar in spirit to wait-freedom [13] or starvation-freedom. Protecting transactions using a single global (and fair) lock ensures local progress. Indeed, transactions that execute sequentially encounter no conflicts: a TM can easily guarantee that each of those transactions commit. Yet, such a TM would not be *resilient*. A resilient TM is, intuitively, one that does not force transactions to wait for each other, thus allowing them to make progress independently.

There are two main problems with non-resilient TMs. First, a transaction that acquires a global lock and gets suspended for a long time, e.g., due to preemption, page faults, or I/O, will prevent all other transactions from making any progress. Even worse, if the transaction enters an infinite loop and keeps running forever, e.g., due to a bug in the program or malicious behavior of some process, then no transaction will ever commit thereafter. Those issues may be important for many applications.

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

For instance, a time-critical transaction might need to commit as soon as possible, and so its progress should not be hampered by a low-priority transaction that is suspended or in the middle of some long computation.

Second, if applications are to scale to multi-core processors, Amdahl's Law recommends to reduce their sequential parts. But if transactions wait for each other, then they contribute to those sequential portions of users' programs, hampering scalability. Finer-grained locking can alleviate this problem, allowing more transactions to execute in parallel, but does not address the issue completely. Resilient TM implementations, which allow transactions to make progress independently, may thus provide better performance on future hardware with a big number of computing cores. Indeed, lock-free implementations of concurrent data structures, which could be considered inefficient when compared to lock-based ones on single-processor systems, have already entered the mainstream (e.g., being included in the Java standard library) because of their better scalability on modern hardware.

The motivation of this work is the question of what liveness property can be ensured by a TM beyond resilience, i.e., when transactions do not wait for each other.

1.2 Transaction failures

The classical way of modeling shared-memory systems in which processes can make progress independently, i.e., without waiting for each other, is to consider asynchronous systems in which processes can *crash*. When a process crashes, it does not perform any further action. Because of asynchrony, processes cannot detect crashes—they cannot distinguish a crashed process from a one that is just very slow or delayed. Therefore, any TM implementation that is resilient to crashes will, in a real system, allow transactions to make progress even if other transactions are suspended for a long time. (Recall that transactions are executed by processes; hence, if a process crashes while executing a transaction, so does the transaction.)

It is not obvious, however, that TMs that can ensure progress despite transaction crashes would also tolerate transactions that keep executing operations on shared data without ever attempting to commit. We call these *parasite* transactions. A priori, when the maximum transaction length is not bounded, parasite transactions can be considered more harmful than crashed ones, because they keep using resources and may cause other transactions to abort. On the other hand, parasite transactions can also be thought of as easier to deal with than crashed ones, because a crashed process cannot actively cooperate and, e.g., be requested to clean up its data or release some of its locks. It is thus desirable to consider two kinds of *failures* of transactions independently: crashes of transactions (i.e., of processes that execute those transactions) and parasite transactions.

This paper is the first formal study of the overall liveness of TM implementations:[1] we ask how much liveness a TM implementation can guarantee in an asynchronous system with transaction failures, and make a first step towards answering this question.

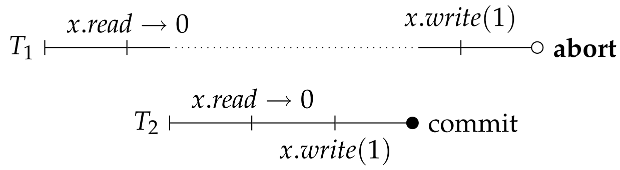

Figure 1: An illustration of why ensuring local progress is impossible in an asynchronous system when transactions can crash. The depicted scenario can repeat infinitely many times, and so transaction T_1 might never be able to commit.

1.3 Challenge

To illustrate the challenge underlying handling faulty (i.e., crashed or parasite) transactions, consider the following example, depicted in Figure 1. Consider two processes, p_1 and p_2, that execute transactions T_1 and T_2, respectively. Transaction T_1 reads value 0 from a shared variable x and then gets suspended for a long time. Then, a transaction T_2 also reads value 0 from x, and attempts to write value 1 to x and commit. Because of asynchrony, processes (and their transactions) can be arbitrarily delayed. Hence, process p_2 does not know whether T_1 has crashed or is just very slow, and so, in order to ensure progress of transaction T_2, p_2 might eventually commit T_2. But then, if transaction T_1 attempts to write value 1 to x, T_1 cannot commit, as this would violate the correctness of the TM (opacity [11] or even serializability [20, 4]). A similar situation can arise if T_1 keeps repeatedly reading variable x instead of being suspended. If the maximum length of a transaction is not known, process p_2 cannot say whether T_1 is a parasite transaction or not, and thus may eventually commit T_2, forcing T_1 to be aborted. A situation in which transaction T_1 has to abort can repeat any number of times, which suggests that implementing local progress with crashed or parasite transactions is impossible. We show in this paper that this is indeed the case.[2]

The question whether a given liveness property, such as local progress, can be implemented by a TM in a system in which transactions can crash and/or be parasite

[1] The *progress* properties of obstruction-free and lock-based TMs, formalized in [10, 12], are, strictly speaking, safety properties. They describe when precisely a particular (single) transaction must commit, assuming certain liveness (namely wait-freedom) of individual operations of this transaction.

[2] It has been pointed out in [8] that ensuring a property analogous to local progress is possible with transaction failures. However, the (informal) model considered in [8] is fundamentally different than ours, in that it allows a TM to control the user application, and, e.g., re-execute portions of a program that uses the TM or simulate an execution of a given transaction, thus enabling the TM to predict what the transaction may do. We believe that this model is too strong.

is, clearly, interesting. Even more interesting, however, is to ask what is the *strongest* liveness property that a TM can ensure despite transaction failures. To answer such a question, one needs a theoretical framework that allows expressing and comparing TM liveness properties in a simple way.

1.4 Contributions

Our first contribution is a formal framework for reasoning about TM liveness. Roughly speaking, we define a TM liveness property to be a function that, for every set of concurrent transactions, specifies which transactions from this set must eventually commit. This definition can capture "high-level" TM liveness properties—those that do not depend on implementation details of a given TM, the structure of transactions, or the exact interleaving of steps of processes. We will motivate this model in Section 3. It is worth noting, however, that all liveness properties that are ensured by current TMs, such as OSTM [9], DSTM [14], RSTM [17], NZTM [22], TL2 [5], TinySTM [7], and SwissTM [6], can be defined in our framework, as well as many TM liveness properties that are thought of as useful in practice (e.g., local progress or priority-based properties).

We then identify, within our framework, two partially overlapping classes of TM liveness properties: *nonblocking* and $(n-1)$-*prioritizing* properties, where n is the number of processes in the system. Intuitively:

1. A TM liveness property is nonblocking if it ensures progress of any transaction that runs alone, i.e., without any concurrent non-faulty transactions.

2. A TM liveness property is $(n-1)$-prioritizing if it specifies, for some execution involving infinitely many transactions, a set of *at most* $n-1$ transactions that have *high priority*. The property then requires that, in this specific execution, at least one of the high-priority transactions must commit (unless all of them are faulty).

For example, local progress is an $(n-1)$-prioritizing property because in any execution involving any transaction T, the single-element set $\{T\}$ is indeed a set of high-priority transactions—local progress guarantees that if T is not faulty then T will eventually commit. Consider also TM liveness properties $L(1), L(2), \ldots$, defined as follows. Assume that transactions are assigned unique identifiers. Property $L(1)$ requires that, in every execution, the non-faulty transaction with the lowest id eventually commits. Similarly, $L(2)$ requires that one of the two non-faulty transactions with the lowest ids eventually commits. And so on. All properties $L(1), \ldots, L(n-1)$ are $(n-1)$-prioritizing, while $L(n), L(n+1), \ldots$, are not. However, a property that simply says that *some* non-faulty transaction, or some arbitrary k transactions, must eventually commit is *not* $(n-1)$-prioritizing because it treats all transactions equally—it does not specify any high-priority ones. Therefore, $(n-1)$-prioritization is all about

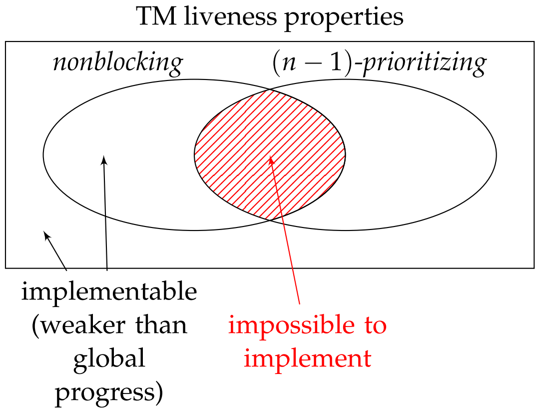

Figure 2: Summary of the results

choice—how constrained is a TM when it must choose which of the concurrent transactions to abort (as in the example from Figure 1).

We then prove the following results, summarized in Figure 2:

1. Every TM liveness property that is both nonblocking and $(n-1)$-prioritizing, e.g., local progress, is *impossible* to implement in a system with crashed transactions *or* parasite transactions.

2. Every TM liveness property that is *not* $(n-1)$-prioritizing can be implemented in a system with crashed *and* parasite transactions.

In order to prove (2), we first identify a TM liveness property that we call *global progress*. Global progress—analogous to lock-freedom in shared-memory implementations—guarantees, intuitively, that at any point in time *some* transaction makes progress. Then, we prove that:

1. Every TM liveness property that is not $(n-1)$-prioritizing is *weaker* than global progress. That is, every TM implementation that ensures global progress ensures also any other non-$(n-1)$-prioritizing TM liveness property.

2. Global progress can be implemented in a system with faulty transactions.

There are indeed TM implementations that ensure global progress in a system with faulty transactions, e.g., OSTM [9]—however, we do not know whether this has been formally proved. We give then in this paper, for completeness, a TM algorithm that (provably) ensures global progress.

Our results imply that global progress is the strongest nonblocking TM liveness property that can be implemented in a system with faulty transactions—a result interesting in its own right. Hence, TM implementations such as OSTM and the TM we show in this paper can be thought of as optimal with respect to liveness, at least within our framework for expressing TM liveness properties.

To summarize, this paper is the first step towards determining how much liveness a TM implementation can ensure in an asynchronous system. We provide a theoretical framework for reasoning about TM liveness properties, and then devise the precise boundary between those TM liveness properties that are implementable in a system with crashed or parasite transactions, and those ones that are impossible to implement in such a system.

1.5 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, gives several examples of TM liveness properties (some ensured by existing TM implementations), and defines the notions of nonblocking and $(n-1)$-prioritizing properties. Section 4 proves that it is impossible to implement any nonblocking $(n-1)$-prioritizing TM liveness property in a system with crashed or parasite transactions. Section 5 shows how to implement any non-$(n-1)$-prioritizing TM liveness property in a system with any faulty transactions. Finally, Section 6 discusses our results and provides some 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 *read* and *write* operations on *transactional variables* (or *t-variables*, for short). Let x be any t-variable and T_i be any transaction. We say that T_i *reads (value v) from x* if T_i executes operation *read* on x, and is returned value v from this operation. We say that T_i *writes (value v) to x*, if T_i executes operation $write(v)$ on x.

Transaction T_i may also issue special operations: $tryC(T_i)$ and $tryA(T_i)$, which are requests to, respectively, commit or abort T_i. Operation $tryC(T_i)$ returns value C_i if committing T_i has been successful, or A_i if T_i has been aborted (committing T_i has failed). Operation $tryA(T_i)$ always returns value A_i.

The special value A_i ($i = 1, 2, \ldots$) can also be returned by any operation executed on any t-variable by transaction T_i. Whenever T_i is returned value A_i from any operation (including $tryC(T_i)$) except for $tryA(T_i)$, we say that T_i has been *forceably aborted*—indeed, T_i is then forced by the TM to abort even though T_i did not request that (by invoking $tryA(T_i)$).

When a transaction T_i is forceably aborted, the operations of T_i on t-variables are rolled back by the TM. Then, T_i may retry its computations. Clearly, each time T_i is forceably aborted, T_i may perform different operations because T_i may observe different states of t-variables.

When a transaction T_i returns value C_i from operation $tryC(T_i)$, or value A_i from operation $tryA(T_i)$, we say that T_i is *completed*. A completed transaction does not perform any further actions. A transaction that is not (yet) completed is called *pending*.

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 an *abort event* of transaction T_k.

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

The TM algorithm is executed by the same processes that execute transactions. That is, whenever a process p_i invokes an operation on any t-variable, or an operation $tryA(T_k)$ or $tryC(T_k)$, within a transaction T_k, p_i follows the TM algorithm by executing corresponding steps until p_i returns from the operation.

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 processes, in a given run of an application. We assume here that every event or step e can be assigned a unique point in time when e was executed. Hence, all events and steps in a given run can be indeed 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.

Let T_i be any transaction in history 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 *active* at t, if (1) T_i has started by t, and (2) either (a) the latest event of T_i that is executed before t is *not* a commit or abort event of T_i, or (b) some event of T_i is executed after t in H.

Let T_i and T_k be any two transactions in history H. We say that T_i *precedes* T_k (in H), if (1) there exists a time after which T_i is never active in H, and (2) the last commit or abort 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).

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 completed, then no event or step follows a response event of operation $tryA(T_k)$ or a commit event of T_k in $H|T_k$.

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*,[3] denoted by $T_k^1, \ldots, T_k^m$, that represent the operations executed by T_k during subsequent retries of T_k. That is, T_k^1 is the subsequence of $H|T_k$ from the first event of T_k until the first commit or abort event of T_k (if any), T_k^2 is the subsequence of $H|T_k$ from the first event of T_k issued after the first abort event of T_k until a commit event or the second abort event of T_k (if any), and so on.

We will apply the same notation and terminology to sub-transactions as defined for transactions. Note that every sub-transaction of any transaction T_k, except the last sub-transaction of T_k, must be pending and forceably aborted.

Correctness condition We assume that every TM implementation M ensures *opacity* [11]. Intuitively, this means that in every history H of M, every sub-transaction (of every 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.

Crashed, parasite and correct 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 say that a transaction T_k crashes in a history H, if the process executing T_k in H crashes at some time at which T_k is active in H.

Let H be any history. We say that a transaction $T_k \in H$ is *infinite*, if sub-history $H|T_k$ is infinite, i.e., if T_k executes infinitely many steps or events in H. Clearly, there can be at most n infinite transactions in H.

Intuitively, a *parasite* transaction is a transaction that keeps executing operations but, from some point in time, never attempts to complete (by invoking operation *tryC* or *tryA*). Consider any history H and any infinite transaction T_k in H. If the last sub-transaction of T_k executes infinitely many operations, then T_k is clearly parasite. Indeed, T_k gets to execute infinitely many operations without being forceably aborted, but T_k does not complete. If T_k invokes operation $tryC(T_k)$ infinitely many times, then T_k is clearly *not* parasite. Consider, however, a situation in which T_k is infinite and does not invoke $tryC(T_k)$ infinitely many times, but T_k is either blocked inside some operation infinitely long or T_k is forceably aborted infinitely many times. Then, looking just at history H, we cannot say whether T_k is parasite, or T_k is simply starving. Indeed, since the TM did not allow T_k to execute infinitely many operations without any forceable abort, and since the maximum number of operations of a given transaction is not known, we do not know whether T_k, if given a chance, would eventually attempt to complete.

Therefore, for every infinite history H (of any TM implementation) we specify a set $Parasite(H)$ of *parasite* transactions, such that, for every transaction T_k in $Parasite(H)$, T_k is an infinite transaction in H, and T_k does not invoke operation $tryC(T_k)$ infinitely many times in H. We assume that the maximum number of operations a single sub-transaction can execute is not known. That is, a TM can reliably determine if a transaction T_i is parasite only by letting T_i execute infinitely many operations in its (last) sub-transaction.

A system S is called *fault-prone* if S is crash-prone or S can have parasite transactions.

Let H be any history. We say that a transaction $T_k \in H$ is *correct* in H if either (1) T_k is completed, or (2) sub-history $H|T_k$ is infinite, but T_k is not a parasite transaction (i.e., $T_k \notin Parasite(H)$).

3 TM Liveness

Intuitively, a TM liveness property describes which of the transactions in a history H have to be (eventually) completed. 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:

1. Property L should be indeed a *liveness* property. That is, L can be violated only in infinite histories, and only by transactions that are pending in those histories. In particular, every history in which all transactions are completed must ensure L.

2. Property L can only restrict *correct* transactions. Indeed, progress can be ensured only for those transactions that execute sufficiently many steps (e.g., do not crash), and that invoke operation *tryC* sufficiently many times (or invoke operation *tryA* once).

We also focus in this paper on high-level TM liveness properties, which do not depend on implementation details of a TM or on conflicts between transactions. Those properties can be ensured by many TMs of different internal design, and so relying on those properties in users' applications should not hamper the portability of those applications between TM implementations. Moreover, ensuring progress regardless of what conflicts transactions encounter is important because conflicts are often unavoidable. This is especially the case for *false* conflicts, which are caused by the internal mechanisms of a TM and thus are even more difficult to avoid at the application level.

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

There are indeed progress properties that depend on conflicts between transactions, e.g., strong and weak progressiveness [12]. Those are safety properties that specify when precisely a TM is allowed to forceably abort a transaction. Often, however, a high-level TM liveness property can be ensured together with a progress property within the same TM implementation. For example, a TM may guarantee that transactions are forceably aborted only if they have encountered a conflict since their last forceable abort (if any), while ensuring global progress for all transactions, even in high-contention scenarios. TM liveness properties can also easily be combined with various heuristics that aim at reducing contention between transactions. This is because a TM liveness property guarantees *eventual* progress. A TM is thus free to forceably abort or delay a transaction T_i many times, e.g., in order to reduce contention, even if the TM liveness property ensures progress for T_i. Transaction T_i must be able to complete, but this can happen after some, possibly long, time.

In principle, a TM liveness property can be a function of some characteristics of a transaction. For example, a TM could give higher priority to transactions that access small number of t-variables, or that write some important data. However, giving priorities to transactions is rather an application-specific task. Hence, we assume that those priorities are encoded in transaction identifiers from set $\mathcal{T}$, which are given to transactions by an application, and TM liveness properties are functions of those identifiers.

In the following sections, we give a definition of the notion of a TM liveness property and illustrate it with a series of examples. Then, we define the different classes of TM liveness properties that we consider in this paper.

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 active. (Time t' can be thought of as the next quiescence time after t.) Consider the set Q of transactions in H that are active at some time between t and t'. Observe first that all transactions in Q are directly or transitively concurrent (we formalize this notion in the next paragraphs). We will call the transactions in set Q a *concurrent group* at time t.

Roughly speaking, a TM liveness property specifies which *correct* transactions from any concurrent group Q in a history H must be completed. We define a TM liveness property in the following way:

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 $S \in L(C)$.*

Intuitively, if Q is the set of correct transactions in the concurrent group at some time t in a history H, and L is a TM liveness property, then $L(Q)$ is a set of subsets Q_1, Q_2, ... of Q. In order to ensure L, all transactions from *some* set $Q_m \in L(Q)$ must be completed in H. (Transactions that are not in Q_m may be either completed or pending.)

More formally, let H be any history and t be any point in time. We denote by $Concurr_H(t)$, and call *concurrent group* at time t, the minimal set C of transactions defined recursively in the following way:

1. If a transaction $T_i \in H$ is active at time t, then $T_i \in Concurr_H(t)$, and
2. If a transaction $T_k \in H$ is active at some time after t and is concurrent to some transaction in $Concurr_H(t)$, then $T_k \in Concurr_H(t)$.

Let H be any history and Q be any set of transactions. We denote by $Correct_H(Q)$ the set of those transactions in Q that are correct in H. We denote by $Completed_H(Q)$ the set of those transactions in Q that are completed in H.

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

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

In order to compare any two TM liveness properties L and L', we define when L is *weaker* than L'. The "weaker than" relation is a partial order on the set of all TM liveness properties. If L is weaker than L', then any TM implementation that ensures L' also ensures L.

Definition 4 *Let L and L' be any two TM liveness properties. We say that L is* weaker *than L' (and L' is* stronger *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. We prove that our definitions of those properties, expressed within our formal framework, do indeed capture the common intuition behind those properties (Theorems 5, 6, and 7). We also give examples of TM implementations that ensure those properties, possibly under some assumptions (e.g., crash-free system, or no parasite transactions).

Local progress Intuitively, a TM implementation M ensures *local 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 completes. More formally, local progress is the function

$$L_l(C) = \{C\}.$$

Every TM liveness property is weaker than L_l.

As we discussed in the introduction, implementing a TM that guarantees local progress in any fault-prone system is impossible. That is, local progress inherently requires some form of indefinite blocking of transactions

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

Theorem 5 *An infinite history H ensures L_l if, and only if, every correct transaction in H is completed in H.*

Proof. ($\Rightarrow$) Let H be any history that ensures L_l and T_k be any correct transaction in H. Let t be any time at which T_k is active in H, and $C = \mathit{Correct}_H(\mathit{Concurr}_H(t))$. Clearly, T_k must be in set C. Because $L_l(C) = \{C\}$, $\mathit{Completed}_H(C)$ must be the entire set C, and so $T_k \in C$ must be completed in H.

($\Leftarrow$) Let H be any history in which every correct transaction is completed. Let t be any time and $C = \mathit{Correct}_H(\mathit{Concurr}_H(t))$. Because every transaction in C is completed, $\mathit{Completed}_H(C) = C \in L_l(C)$. ☐

Global progress Intuitively, a TM implementation M ensures *global progress* (analogous to *lock-freedom* for shared-memory objects), if in every infinite history of M, in which there is a pending correct transaction, there are infinitely many completed transactions. More formally, global progress is the function

$$L_g(C) = \{\{T_{i_1}\}, \{T_{i_2}\}, \dots\},$$

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 [9]. We give a simple TM implementation that guarantees global progress in Section 5. 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 [7], or SwissTM [6] allow livelock situations—scenarios in which two concurrent correct transactions are pending forever).

Theorem 6 *An infinite history H ensures L_g if, and only if, whenever there is a pending correct transaction in H, then infinitely many transactions are completed in H.*

Proof. ($\Rightarrow$) Let H be any infinite history that ensures L_g. Assume that there is a pending correct transaction T_k in H. By contradiction, assume that after some time t no transaction completes. Because T_k is pending, set $Q = \mathit{Correct}_H(\mathit{Concurr}_H(t'))$, where $t' > t$, contains at least transaction T_k. Hence, by L_g, some transaction from Q must be completed in H—a contradiction.

($\Leftarrow$) Let H be any infinite history. If H has no pending correct transaction, then H trivially ensures L_g. Assume then that H contains a pending correct transaction T_k and there are infinitely many completed transactions in H. But then, for every time t, set $Q = \mathit{Correct}_H(\mathit{Concurr}_H(t))$ contains either (1) only completed or incorrect transactions, or (2) some pending correct transactions and infinitely many completed transactions. In both cases property L_g is ensured. ☐

Solo progress Intuitively, a TM implementation M ensures *solo progress* (analogous to *obstruction-freedom* for shared-memory objects), 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". Parasite 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 transactions. In a system without parasite 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:

$$L_s(C) = \begin{cases} \{C\} & \text{if } |C| = 1 \\ \{\emptyset\} & \text{otherwise.} \end{cases}$$

In a crash-prone system without parasite transactions, solo progress is ensured by TM implementations such as DSTM [14], RSTM [17] (with its nonblocking backend), or NZTM [22]. In a crash-free system without parasite transactions, solo progress is ensured by most (if not all) lock-based TM implementations, e.g., TL2 [5], TinySTM [7], or SwissTM [6]. (In fact, the progress semantics of those TMs, as formalized in [12], is stronger than solo progress in a crash-free system.) However, only lazy-acquire TMs, such as TL2, ensure solo progress with parasite transactions.

Theorem 7 *An infinite history H without parasite transactions ensures property L_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 completes.*

Proof. ($\Rightarrow$) Let H be any infinite history without parasite transactions that ensures property L_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 $\mathit{Correct}_H(\mathit{Concurr}_H(t)) = \{T_k\}$. Hence, because $L_s(\{T_k\}) = \{\{T_k\}\}$, T_k must be completed in H—a contradiction.

($\Leftarrow$) Let H be any infinite history without parasite transactions. If every correct transaction is completed in H, then H trivially ensures L_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 after T_k starts, set $\mathit{Correct}_H(\mathit{Concurr}_H(t))$ contains some transaction other than T_k, and so L_s is ensured. ☐

Priority-based properties Local progress, global progress, and solo progress give the same guarantees to all transactions, regardless their identifiers. We can, however, encode some priority scheme into those identifiers, thus giving preference to some transactions. For instance, let $\ll$ be any total order on set $\mathcal{T}$. Let us define the following TM liveness property:

$$L_{\ll,1}(C) = \{T_k\},$$

where $T_k \ll T_i$ for every transaction $T_i \in C$ different than T_k. Intuitively, $L_{\ll,1}$ ensures progress for the transaction with the lowest transaction id (according to order $\ll$) from a given concurrent group.

Consider also the following TM liveness property:

$$L_{\ll,n}(C) = \begin{cases} \{C\} & \text{if } |C| \leq n \\ \{\{T_{\sigma_1}\}, \ldots, \{T_{\sigma_n}\}\} & \text{otherwise,} \end{cases}$$

where $T_{\sigma_1} \ll \ldots \ll T_{\sigma_n}$, and $T_{\sigma_n} \ll T_i$ for every transaction $T_i \in C$. Intuitively, $L_{\ll,n}$ ensures progress for one of the n transactions with the lowest id from a given concurrent group.

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:

Definition 8 *We say that a TM liveness property L is* nonblocking *if, for every transaction $T_i \in \mathcal{T}$, $L(\{T_i\}) = \{\{T_i\}\}$.*

Properties that are not nonblocking are called *blocking*. Local progress, global progress, an solo progress, as well as properties $L_{\ll,1}$ and $L_{\ll,n}$ for every total order $\ll$ on set $\mathcal{T}$, are all nonblocking TM liveness properties.

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., $Correct_H(Concurr_H(t)) = C$ at some time t), then for history H to ensure L at least one of the transactions in set P must be completed in H. In a sense, P is a set of transactions with higher priority, and one of those transactions has to complete in the (single) scenario described by set C.

More formally, let L be any TM liveness property.

Definition 9 *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 $L(C)$, $P \cap S \neq \emptyset$.*

Local progress and property $L_{\ll,1}$ (for any total order $\ll$ on set $\mathcal{T}$) are $(n-1)$-prioritizing. Global progress, solo progress, and property $L_{\ll,n}$ are not $(n-1)$-prioritizing.

4 Impossibility Result

In this section, we prove that TM liveness properties that are nonblocking and $(n-1)$-prioritizing, such as local progress, are impossible to implement in any fault-prone system, i.e., in a system with crashes and/or parasite transactions (Theorem 11). We start by proving the following lemma, which says, intuitively, that a process executing infinitely many transactions can block progress of all other processes, if the TM implementation ensures any nonblocking TM liveness property.

Lemma 10 *For every TM implementation M that ensures any nonblocking TM liveness property in any fault-prone system, 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 $Correct_H(Concurr_H(t)) = C$ 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 in any fault-prone system S. Let P and C be any sets such that $P \subset C \subseteq \mathcal{T}$, $|C| = \infty$, and $|P| = n-1$. (Recall that n is the number of processes). Denote by σ any one-to-one function from the set of natural numbers to set C. For simplicity of notation, but without loss in generality, we will assume in the following that $\sigma(k) = k$ for $k = 1, \ldots, n-1$ and that $P = \{T_1, \ldots, T_{n-1}\}$.

We consider two cases: when system S is crash-prone and when S can have parasite transactions. In each case we show an infinite history H in which every transaction from set P is correct, pending, and concurrent to all transactions from set C.

Case 1: S **is a crash-prone system** Consider a history H of M generated in the following execution (initially, $k = n$; x is some t-variable initialized to 0):

1. Transactions $T_1, \ldots, T_{n-1}$ read x one by one, i.e., each transaction T_i, $i = 2, \ldots, n-1$, invokes operation *read* on x after T_{i-1} returns from its operation *read* on x.
2. Transaction $T_{\sigma(k)}$ reads some value v from t-variable x and writes value $1 - v$ to x. Then, $T_{\sigma(k)}$ attempts to commit, i.e., executes operation *tryC*$(T_{\sigma(k)})$. Whenever $T_{\sigma(k)}$ is forceably aborted, $T_{\sigma(k)}$ retries the same operations until it commits. No transaction executes steps concurrently to $T_{\sigma(k)}$.
3. Those transactions from set $\{T_1, \ldots, T_{n-1}\}$ that were not forceably aborted in step 1 write value $1 - v$ to x and attempt to commit one by one. If all of them are forceably aborted, go back to step 1 with $k \leftarrow k + 1$.

Assume that every transaction T_i, $i = 1, \ldots, n-1$, is executed by process p_i, and all transactions $T_{\sigma(k)}$, $k = n, n+1, \ldots$, are executed by process p_n in H. Assume also that no transaction crashes in H.

We prove first that the above algorithm cannot be blocked in step 1 or 3, i.e., that no transaction T_i, $i = 1, \ldots, n-1$, can be blocked by M inside its *read* or *write* operation on x, or inside operation *tryC*(T_i), infinitely long. Assume otherwise—that some transaction T_i, $1 \leq$

$i \leq n-1$, invokes an operation *op*, executes infinitely many steps, but never receives a response event from *op*. Consider a history H' identical to H except that every transaction T_k, $k = 1, \ldots, n-1$ and $k \neq i$, crashes in H' just before T_i invokes operation *op*. Hence, there is some time t at which T_i is the only correct transaction in H', i.e., $\mathit{Correct}_{H'}(\mathit{Concurr}_{H'}(t)) = \{T_i\}$. Then, because M ensures L and $L(\{T_i\}) = \{\{T_i\}\}$, T_i must be completed in H'. But process p_i, which executes T_i, cannot distinguish between histories H and H', and so T_i cannot be pending in H—a contradiction.

Assume then, by contradiction, that, for some value k, $k \geq n$, transaction $T_{\sigma(k)}$ is pending in H. Then, no transaction T_i, $i = 1, \ldots, n-1$, issues any event or step after the first event of $T_{\sigma(k)}$ in H. Let H' be a history that is identical to H but in which all transactions $T_1, \ldots, T_{n-1}$ crash just before the first event of $T_{\sigma(k)}$. But then, at some time t, $T_{\sigma(k)}$ is the only correct transaction in H', i.e., $\mathit{Correct}_{H'}(\mathit{Concurr}_{H'}(t)) = \{T_{\sigma(k)}\}$. Hence, as M ensures L and $L(\{T_{\sigma(k)}\}) = \{\{T_{\sigma(k)}\}\}$, $T_{\sigma(k)}$ must be completed in H'. But process p_n cannot distinguish between histories H and H', and so $T_{\sigma(k)}$ must also be completed in H—a contradiction.

Finally, assume, by contradiction, that some transaction T_m, $1 \leq m \leq n-1$, commits. Denote by T_m^q the last (committed) sub-transaction of T_m. Let $T_{\sigma(w)}$ be the latest transaction $T_{\sigma(k)}$, $k \geq n$, that precedes T_m^q, and let v_w be the value written to t-variable x by $T_{\sigma(w)}$. Until T_m^q returns from its *read* operation on 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^q must read v_w from x. Then, T_m^q writes value $1 - v_w$ to x and commits.

Consider transaction $T_{\sigma(w+1)}$, which is concurrent to sub-transaction T_m^q of T_m. We already proved that $T_{\sigma(w+1)}$ must be committed in H. Denote by $T^s_{\sigma(w+1)}$ the last sub-transaction of $T_{\sigma(w+1)}$. Both T_m^q and $T^s_{\sigma(w+1)}$ read value v_w from x and write value $1 - v_w$ to x, and no transaction concurrent to T_m^q or $T^s_{\sigma(w+1)}$ writes back value v_w to x. Hence, there is no way to order sub-transactions T_m^q and $T^s_{\sigma(w+1)}$, and so opacity is violated—a contradiction.

Case 2: S is a system with parasite transactions Consider a history H of M generated in the following execution, similar to the one considered in Case 1 (initially, $k = 1$; x is some t-variable initialized to 0):

1. The following steps are executed in parallel by all processes (e.g., in lockstep) until transaction $T_{\sigma(k)}$ commits:
   (a) Every transactions T_i, $i = 1, \ldots, n-1$, repeatedly executes operation *read* on x until T_i gets forceably aborted.
   (b) Transaction $T_{\sigma(k)}$ repeatedly executes operation *read* on x until every transaction T_i, $i = 1, \ldots, n-1$, either gets forceably aborted or returns from at least one *read* operation on x since $T_{\sigma(k)}$ started. Then, $T_{\sigma(k)}$ writes to x value $1 - v$, where v is the value returned by the latest *read* operation of $T_{\sigma(k)}$ on x. Finally, $T_{\sigma(k)}$ attempts to commit, i.e., $T_{\sigma(k)}$ issues operation $\mathit{tryC}(T_{\sigma(k)})$. Whenever $T_{\sigma(k)}$ is forceably aborted, $T_{\sigma(k)}$ retries the same operations.
2. Every transaction T_i, $i = 1, \ldots, n-1$, that has not been forceably aborted in step 1a first finishes its current *read* operation on x (if any), then writes value $1 - v$ to x, and executes operation $\mathit{tryC}(T_i)$. When every transaction T_i is forceably aborted, go to step 1 with $k \leftarrow k + 1$.

Assume that every transaction T_i, $i = 1, \ldots, n-1$, is executed by process p_i, and all transactions $T_{\sigma(k)}$, $k = n, n+1, \ldots$, are executed by process p_n in H. Assume also that $\mathit{Parasite}(H) = \emptyset$. That is, there are no parasite transactions in H.

We prove first that no sub-transaction T_i^m in H can be blocked inside its first *read* operation on x (in step 1a or 1b) infinitely long. Assume otherwise—that some sub-transaction $T_i^m \in H$ invokes its first *read* operation on x, executes infinitely many steps, but never returns from this operation. Then, every transaction in H, including T_i, is pending (by the construction of H), and no transaction T_k in H invokes operation $\mathit{tryC}(T_k)$ or $\mathit{tryA}(T_k)$ infinitely many times. Consider then a history H' that is identical to H but in which all transactions that execute infinitely many steps or events in H', except for T_i, are parasite in H'. Then, T_i is the only correct transaction in H', and so T_i cannot be pending in H'. Hence, because the process executing T_i cannot distinguish between histories H and H', T_i cannot be pending in H—a contradiction.

Assume then, by contradiction, that, for some value k, transaction $T_{\sigma(k)}$ is pending in H. We proved before that every sub-transaction in H always eventually returns from its first *read* operation on x. Hence, $T_{\sigma(k)}$ cannot execute infinitely many *read* operations on x in step 1b without being forceably aborted infinitely many times. Hence, $T_{\sigma(k)}$ does not have to be a parasite transaction. Every transaction T_i, $i = 1, \ldots, n-1$, either executes infinitely many steps while reading x in step 1a, or does not execute any further events after being forceably aborted in step 1a. Hence, T_i can be a parasite transaction. Consider then a history H' identical to H except that all transactions $T_1, \ldots, T_{n-1}$ are parasite in H'. But then, $T_{\sigma(k)}$ is, at some time t, the only correct transaction in H'. Hence, because $L(\{T_{\sigma(k)}\}) = \{\{T_{\sigma(k)}\}\}$, $T_{\sigma(k)}$ cannot be pending in H'. Therefore, because process p_n cannot distinguish between histories H and H', $T_{\sigma(k)}$ also cannot be pending in H—a contradiction.

We proved that the first *read* operation on x of every sub-transaction in H must eventually return. We also proved that no transaction $T_{\sigma(k)}$, $k = n, n+1, \ldots$, can be pending in H, which means that every operation of $T_{\sigma(k)}$

eventually returns a response event. We prove now that every operation of any transaction T_i, $1 \leq i \leq n-1$, must eventually return a response event in H. Assume otherwise—that some transaction T_i invokes an operation *op*, executes infinitely many steps, but never receives a response from *op*. By the construction of history H, there must be then only finitely many transactions in H, and no transaction T_k in H invokes operation $tryC(T_k)$ or $tryA(T_k)$ infinitely many times. Consider then a history H' that is identical to H except that in H' all transactions that execute infinitely many steps in H', except for T_i, are parasite. Hence, eventually, at some time t, T_i is the only correct transaction in H', and so T_i cannot be pending in H'. This means, again, that T_i cannot be pending in H, as process p_i cannot distinguish H from H'—a contradiction.

Finally, we need to prove that no transaction T_i, $i = 1, \ldots, n-1$, commits. Assume otherwise—that some transaction T_m, $1 \leq m \leq n-1$, commits. Denote by T_m^q the last sub-transaction of T_m, i.e., the sub-transaction of T_m that ends with a commit event. Let $T_{\sigma(w)}$ be the latest transaction $T_{\sigma(k)}$, $k = 1, 2, \ldots$, that precedes T_m^q, and v_w be the value written to t-variable x by $T_{\sigma(w)}$. Until T_m^q reads x for the first time (in step 1a), there is no concurrent transaction that writes to x. Hence, because M ensures opacity, T_m^q must read v_w from x in its every read operation on x. Then, T_m^q writes value $1 - v_w$ to x and commits.

Consider transaction $T_{\sigma(w+1)}$, which is concurrent to sub-transaction T_m^q of T_m. We already proved that $T_{\sigma(w+1)}$ must be committed in H. Denote by $T_{\sigma(w+1)}^s$ the last sub-transaction of $T_{\sigma(w+1)}$. Both T_m^q and $T_{\sigma(w+1)}^s$ read value v_w from x and write value $1 - v_w$ to x, and no transaction concurrent to T_m^q or $T_{\sigma(w+1)}^s$ writes back value v_w to x. Hence, there is no way to order sub-transactions T_m^q and $T_{\sigma(w+1)}^s$, and so opacity is violated—a contradiction. $\square$

Theorem 11 *There does not exist any nonblocking $(n-1)$-prioritizing TM liveness property that can be implemented in any fault-prone system.*

Proof. Let L be any nonblocking, $(n-1)$-prioritizing TM liveness property. Because L is $(n-1)$-prioritizing, there exists an infinite set $C \subseteq \mathcal{T}$ and a set $P \subset C$ of size at most $n-1$ such that $P \cap S \neq \emptyset$ for every $S \in L(C)$. Let P' be any subset of C of size $n-1$ that contains all elements in set P. Clearly, $P' \cap S \neq \emptyset$ for every $S \in L(C)$.

By contradiction, assume that there is a TM implementation M that ensures L in some fault-prone system. By Lemma 10, and because L is nonblocking, there exists a history H of M and a time t such that $Correct_H(Concurr_H(t)) = C$ 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 $Completed_H(C) \not\supseteq S$. Hence, history H of M violates L—a contradiction with the assumption that M ensures L. $\square$

5 Ensuring Non-$(n-1)$-Prioritizing TM Liveness Properties

In the previous section, we showed that TM liveness properties that are nonblocking and $(n-1)$-prioritizing are impossible to implement in any fault-prone system. In this section, we prove that every TM liveness property L that is *not* $(n-1)$-prioritizing, regardless of whether L is blocking or nonblocking, can be implemented in every fault-prone system. In fact, we prove that every such property L is weaker than global progress. We show then a TM implementation that implements global progress, and so also any weaker property, in a crash-prone system with parasite transactions.

Note that the OSTM [9] implementation also ensures global progress in any fault-prone system. However, we do not know if this has been formally proved. Hence, for completeness, we give here our own TM algorithm and prove that it indeed ensures opacity and global progress in presence of crashed and parasite transactions. The purpose of the TM we show in this section is only to prove our result—the TM is not meant to be practical or efficient. It is worth noting, however, that proving correctness of even such a simple TM implementation is a technical challenge.

Theorem 12 *Every TM liveness property that is not $(n-1)$-prioritizing is weaker than L_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_g.

Because L is not weaker than L_g, there exists a history H such that H ensures L_g and H does not ensure L. Because H does not ensure L, there is a time t such that, if $C = Correct_H(Concurr_H(t))$, then $Completed_H(C) \not\supseteq S$ for every $S \in L(C)$. Note first that if C is a finite set, then L_g requires that all transactions in C are completed, i.e., that $Completed_H(C) = C$. 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_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 completed 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 $Completed_H(C) = C - P \supseteq C - P'$ is a superset of S—a contradiction. $\square$

An example TM implementation that ensures global progress in a crash-prone system with parasite transactions is shown in Algorithm 1. The intuition behind the algorithm is the following (a proof of correctness is in A). 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-variables and stores those states in the next available slot of array S (lines 7–8). Process p_i searches for

Algorithm 1: A TM implementation that ensures L_g (code for each process p_i; $x_1, \ldots, x_K$ are t-variables 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 shared variables, 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-variable x_m (for $m = 1, \ldots, K$), $C = (1, 1)$, $slot_i = \perp$ (at every process p_i)

1 **upon** *operation op on t-variable* x_m *by transaction* T_k **do**
2   **if** $slot_i = \perp$ **then**
3         $slot_i = 1$;
4   **while** $A[slot_i].\mathit{test\text{-}and\text{-}set} = 1$ **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   **if not** $C.\mathit{compare\text{-}and\text{-}swap}\ ((curr_i, ver_i),\ (slot_i, ver_i + 1))$ **then return** $abort(T_k)$;
15     $A[curr_i].reset()$;
16     $slot_i \leftarrow \perp$;
17   **return** C_k;

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

an available slot s by scanning array A of test-and-set objects[4] (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[5] 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-variable states (the one p_i read at the beginning of sub-transaction T_k^l). Otherwise, p_i releases its own slot.

By proving correctness of Algorithm 1 in Appendix A, we prove thus the following theorem:

Theorem 13 *There is a TM implementation that ensures global progress in every fault-prone system.*

Corollary 14 *Global progress is the strongest nonblocking TM liveness property that can be ensured by any TM in any fault-prone system.*

6 Discussion

This paper is the first step towards determining how much liveness a TM implementation can ensure. We define precisely the notion of a TM liveness property, and show a precise boundary between those TM liveness properties that can be implemented in system with crashed and/or parasite transactions, and those ones that are impossible to implement in such a system. As we pointed out in the introduction, these are preliminary steps towards understanding TM liveness. In the following, we discuss our results, and present some open questions.

Practical perspective Local progress is an important property—it guarantees freedom from starvation of every process in the system. Other practically relevant nonblocking and $(n-1)$-prioritizing properties, besides local progress, may, e.g., guarantee progress for all transactions with certain priority. Our results imply that ensuring any such TM liveness property inherently requires processes to wait for each other. This means, for instance, that ensuring worst-case liveness, such as local progress, may hamper the average-case performance of a TM.

Ensuring local progress without making transactions wait for each other is possible if transactions are static and predefined. That is, if, upon the first event of a transaction T_i, the TM implementation knows exactly which operations, on which t-variables T_i will execute, our impossibility result does not hold. However, assuming static transactions may often be too limiting for an application.

Dissecting the "grey area" The class of TM liveness properties that are $(n-1)$-prioritizing and blocking is a "grey area". It contains properties that are implementable, and properties that are impossible to implement in a system with faulty transactions.

There are many TM liveness properties in this "grey area", which are $(n-1)$-prioritizing and nonblocking if

[4] 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.

[5] A compare-and-swap object implements an operation $\mathit{compare\text{-}and\text{-}swap}(v, v')$ 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.

the set of all transactions is restricted to some infinite subset of set $\mathcal{T}$. It is straightforward to see that all those properties are impossible to implement in any fault-prone system. It is also easy to show that all TM liveness properties that guarantee progress only for transactions from some *finite* subset of set $\mathcal{T}$ are weaker than global progress, and are thus implementable in any fault-prone system. Those two groups of properties probably include all properties from the "grey area" that can potentially be of any practical relevance. Hence, we believe that dissecting the class of $(n-1)$-prioritizing and blocking properties further makes little sense.

Complexity An interesting future direction is related to complexity. Indeed, we showed which TM liveness properties are possible to implement in a system with crashes and/or parasite transactions. However, the inherent cost, in terms of time and space complexity, of ensuring all those properties might not necessarily be the same, and this cost can also differ depending on how many transactions can crash or be parasite in a given system. 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 *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing (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 *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006.

[6] A. Dragojević, R. Guerraoui, and M. Kapałka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[7] P. Felber, T. Riegel, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[8] C. Fetzer. Robust transactional memory and the multicore system model. Presentation at the DISC'09 workshop What Theory for Transactional Memory? (WTTM), 2009.

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

[10] R. Guerraoui and M. Kapałka. On obstruction-free transactions. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[11] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[12] R. Guerraoui and M. Kapałka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

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

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

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

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

[17] 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 *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.

[18] 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 *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[19] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

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

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[22] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: nonblocking zero-indirection transactional memory. In *2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2007.

[23] J. Vitek, S. Jagannathan, A. Welc, and A. Hosking. A semantic framework for designer transactions. In *Proceedings of the European Symposium on Programming (ESOP)*, 2004.

A Proof of Correctness of Algorithm 1

Denote Algorithm 1 by M. We prove that M ensures opacity and global progress in a crash-prone system with parasite transactions.

Opacity Let H be any history of M. Observe first that each object $A[s]$ acts as a lock for the shared variables $S[s][1,\ldots,K]$. 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 variable $S[s][1,\ldots,K]$ until T_k^l executes line 15 or line 19. Hence, variables $S[s][1,\ldots,K]$ 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-variables (reading $S[c_k^l][1,\ldots,K]$ in line 8), and a sequence of one or more *write* operations on every t-variable x_m (writing $S[slot_i][m]$ in line 8 and line 10). Hence, T_l^k first reads from every t-variable x_m, then writes to every t-variable x_m, and then writes to some t-variables. Without loss in generality, we can assume that each value written to a t-variable 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 [11].

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 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 $v_k^m \le v_j^l$, 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 $v_k^m \le v_j^l$, 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 [11].)

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 variables $S[c'][1,\ldots,K]$ 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. □

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 variable $S[c_k^l][m]$ 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 variable $S[c_z^x][m]$ 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 completes 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 forceably 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 completes 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 be forceably aborted—a contradiction.