On the Liveness of Transactional Memory

Victor Bushkov
EPFL, IC, LPD
victor.bushkov@epfl.ch

Rachid Guerraoui
EPFL, IC, LPD
rachid.guerraoui@epfl.ch

Michał Kapałka
EPFL, IC, LPD
michal.kapalka@epfl.ch

ABSTRACT

Despite the large amount of work on Transactional Memory (TM), little is known about how much liveness it could provide. This paper presents the first formal treatment of the question. We prove that no TM implementation can ensure local progress, the analogous of wait-freedom in the TM context, and we highlight different ways to circumvent the impossibility.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements / Specifications

General Terms

Theory

Keywords

Concurrent programming, Liveness, Transactional memory

1. INTRODUCTION

Transactional memory (TM) [10, 13, 20] is a concurrency control paradigm that aims at simplifying concurrent programming. Each sequential process (or thread[1]) of an application performs operations on shared data within a *transaction* and then either commits or aborts the transaction. If the transaction is committed, then the effects of its operations become visible to subsequent transactions; if it is aborted, then the effects are discarded. Transactions are viewed as a simple way to write concurrent programs and hence leverage multicore architectures. Not surprisingly, a large body of work has been dedicated to implementing the paradigm and reducing its overheads.

[1]The technical difference between threads and processes is not important for the theoretical results of the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'12, July 16–18, 2012, Madeira, Portugal.
Copyright 2012 ACM 978-1-4503-1450-3/12/07 ...$10.00.

To a large extent, however, setting the theoretical foundations of the TM concept has been neglected. Indeed, correctness conditions for TMs have been proposed in [9, 15, 4] and programming language level semantics of specific classes of TM implementations have been determined, e.g., in [1, 16, 17, 18]. All those efforts, however, focused solely on *safety*, i.e., on what TMs *should not do*. Clearly, a TM that ensures only a safety property can trivially be implemented by aborting all operations. To be meaningful, a TM has to ensure a *liveness* property [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 [14]. A widely studied such property is *wait-freedom* [11]. It ensures, intuitively, that *every* process invoking an operation eventually returns from this operation, even if other processes crash. It is the ultimate liveness property in concurrent computing as it ensures that every process makes progress.

In a transactional context, requiring such a property alone would however not be enough to ensure any meaningful progress: processes of which all transactions are *aborted* might be satisfying wait-freedom but would not be making real progress. To be meaningful, a TM liveness property should ensure transaction *commitment*, beyond operation *termination*.

One would expect from a TM that every process that keeps executing a transaction (say keeps retrying it in case it aborts) eventually commits it—a property that we call *local progress* and that is similar in spirit to wait-freedom. Not satisfying this property means that some transaction, even when retried forever, might never commit.

In fact, a TM implementation that protects transactions using a single fair global lock could ensure local progress: such a TM would execute all transactions sequentially, thus avoiding transaction conflicts. Yet, such a TM would force processes to wait for each other, preventing them from progressing independently. A process that acquires a global lock and gets suspended for a long time, or that enters an infinite loop and keeps running forever without releasing the lock, would prevent all other processes from making any progress. This would go against the very essence of wait-freedom. Hence, to be really meaningful a TM liveness property should enforce some "independent" progress.

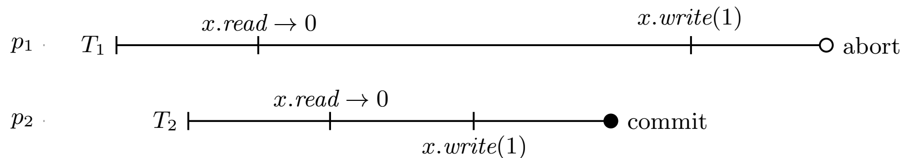

Figure 1: An illustration of the difficulty of ensuring local progress. The scenario can repeat infinitely many times preventing transaction T_1 from ever committing.

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 be arbitrarily slow, including failing by *crashing*. A TM implementation that is resilient to crashes enables the progress of a process even if other processes are suspended for a long time. In the same vein, one should also ensure progress in the face of *parasitic* processes—those that keep executing transactional operations without ever attempting to commit. These model long-running processes whose duration cannot be anticipated by the system, e.g., because of an infinite loop.

To illustrate the underlying challenges, consider the following example, depicted in Figure 1. Two processes, p_1 and p_2, execute transactions T_1 and T_2, respectively. Process p_1 reads value 0 from a shared variable x and then gets suspended for a long time. Then, process p_2 also reads value 0 from x, writes value 1 to x, and attempts to commit. Because of asynchrony, the processes can be arbitrarily delayed. Hence, the TM does not know whether p_1 has crashed or is just very slow, and so, in order to ensure the progress of process p_2, the TM might eventually allow process p_2 to commit T_2. But then, if process p_1 writes value 1 to x and attempts to commit T_1, the TM cannot allow process p_1 to commit, as this would violate safety. A similar situation can occur in the case of parasitic processes, say if p_1 keeps repeatedly reading from variable x. If the maximum length of a transaction is not known, the TM cannot say whether p_1 is parasitic or not, and thus may eventually allow process p_2 to commit T_2, forcing process p_1 to abort T_1 later.

1.3 Contributions

This paper first introduces the notion of a *TM-liveness* property which specifies, for each infinite execution, which processes should make progress, i.e. which processes commit transactions infinitely often. We formalize this notion by modeling TM implementations as I/O automata and focusing on infinite histories of such automata.

Since safety properties state that some events should not occur and liveness properties state that some events should eventually occur, safety and liveness requirements depend on each other. A safety requirement may make it impossible to guarantee a liveness requirement and vice versa. The question is, under what conditions which safety and liveness properties are impossible to guarantee? We address this question in the TM context by proving an impossibility result which states that no TM implementation can ensure both *local progress* and *opacity* in a fault-prone system, i.e. in a system in which any number of processes can crash or be parasitic. Opacity is the safety property ensured by most TM implementations. It states that every transaction observes a consistent state of the system. Local progress is a TM-liveness property, highlighted above, which states that every correct process, i.e. a process which is not parasitic and does not crash, makes progress. In fact, we prove a more general result stating that no TM implementation can ensure any safety property that is at least as strong as strict serializability together with the progress of at least two correct processes and any correct process that runs alone.

2. PRELIMINARIES

2.1 Processes and Shared Memory

We assume a classical (see, e.g., [11]) asynchronous, shared memory system of n *processes* $p_1, \ldots, p_n$ that communicate by executing operations on *shared objects* (which represent the shared memory, e.g., provided in hardware). A *shared object* is a higher-level abstraction provided to processes, and implemented typically in software using a set of *base objects*.

For instance, if base objects are memory locations with basic operations such as read, write, and compare-and-swap, then shared objects could be shared data structures such as linked lists or hash tables. If a process p_i invokes an operation *op* on a shared object O, then p_i follows the implementation of O, possibly accessing any number of base objects and executing local computations, until p_i is returned a result of *op*. We assume that processes are sequential; that is, whenever a process p_i invokes an operation *op* on any shared object, p_i does not invoke another operation on any shared object until p_i returns from *op*. Invocations and responses of shared objects operations are called (invocation and response) *events*.

2.2 Transactional Memory

Let K be the set of *process identifiers*, $P = \{p_k | k \in K\}$ be a set of processes, and X be a set of shared objects called *t-variables* ("t-variable" stands for "transactional variable"). The theoretical results of the paper hold for any shared objects which can implement read and write operations. Thus, for presentation simplicity, we focus on t-variables that support *read* and *write* operations. Each t-variable can have values from set V. Let $Inv_k = \{x.write^k(v) | x \in X \text{ and } v \in V\} \cup \{x.read^k | x \in X\} \cup \{tryC^k\}$ be the set of invocation events of process p_k and $Res_k = \{v^k | v \in V\} \cup \{ok^k, A^k, C^k\}$ be the set of response events of process p_k, where $tryC^k$ is a commit request, C^k is a *commit event*, and A^k is an *abort event*. Also, let $Inv = \cup_{k \in K} Inv_k$ and $Res = \cup_{k \in K} Res_k$.

Since a TM is a discrete event system that receives inputs from processes and returns corresponding responses we model behavior of TM implementations using I/O au-

tomata [8]. Formally, an I/O automaton F is a quintuple (St, I, Int, O, St_0, R), where St is a (possibly infinite) set of states, I is a set of input events, Int is a set of internal events, O is a set of output events, $St_0 \subseteq St$ is the set of initial states, $R \subseteq St \times (I \cup O) \times St$ is a transition relation. The sets I, Int, and O are disjoint. An *execution* of automaton F is a (finite or infinite) sequence $s_0 \cdot e_1 \cdot s_1 \cdot e_2 \cdot s_2 \cdot \ldots$ of alternating states and events such that (I) the sequence starts from an initial state $s_0 \in St_0$, (II) for any $i \in \{1, 2, \ldots\}$ we have $(s_{i-1}, e_i, s_i) \in R$, (III) the sequence ends with a state in case the sequence is finite. The longest subsequence of an execution of automaton F such that the subsequence consists of the events from $(I \cup O)$ is called a *history* H of automaton F.

Let F be an I/O automaton such that $I = Inv$ and $O = Res$. Denote by Σ_k a set such that $\Sigma_k = \{x.write^k(v) \cdot ok^k | x.write^k(v) \in Inv_k\} \cup \{x.read^k() \cdot v^k | x.read^k() \in Inv_k$ and $v^k \in Res_k\} \cup \{tryC^k \cdot C^k\} \cup \{e \cdot A^k | e \in Inv_k\}$. Also, let $\Sigma_k^\infty = \Sigma_k^* \cup \Sigma_k^\omega$, where Σ_k^* is the set of all finite sequences over Σ_k and Σ_k^ω is the set of all infinite sequences over Σ_k. Let H be a history over $Inv \cup Res$, we define a *projection* $H|p_k$ of H on process p_k as the longest subsequence of H consisting of events from $Inv_k \cup Res_k$. A history H is *well-formed* iff for every $p_k \in P$ either $H|p_k \in \Sigma_k^\infty$ or $H|p_k \in \Sigma_k^* \cdot Inv_k$ holds, i.e. $H|p_k$ is a sequence of alternating invocation and response events.

We model a *TM implementation* as an I/O automaton $F = (St, I, Int, O, St_0, R)$ such that:

- $I = Inv$ and $O = Res$.
- Every history H of F is well-formed.
- For every $k \in K$, every $e \in Inv_k$, and every history H of F such that $H|p_k \in \Sigma_k^*$, history $H \cdot e$ is a history of F. In other words, every process which is not waiting for a response must be allowed by F to send an invocation event.

Given projection $H|p_k$ of history H of some TM implementation, a *transaction* of p_k in H is a subsequence $T = e_1 \cdot \ldots \cdot e_n$ of $H|p_k$ such that e_1 is the first event in $H|p_k$ or the previous event e_0 in $H|p_k$ is either A^k or C^k, and e_n is either A^k or C^k or the last event in $H|p_k$, and no event in T, except e_n, is A^k or C^k. Transaction T is *committed* (*aborted*) if the last event in T is a commit (abort) event. Given transactions T_1 and T_2 in history H, we say that T_1 *precedes* T_2 in H, denoted by $T_1 <_H T_2$, if T_1 is committing or aborting and the last event of T_1 occurs in H before the first event of T_2. Transactions T_1 and T_2 are *concurrent* if T_1 does not precede T_2 and T_2 does not precede T_1. History H is *sequential* if no two transactions in H are concurrent to each other.

Processes communicate with each other only through a TM implementation by invoking concurrently requests (read, write, and commit requests) and receiving corresponding responses from the implementation. Processes send commit requests to the TM implementation that decides which transactions should be committed or aborted. To reduce contention between transactions, a TM implementation may use a logically separate module called a contention manager. A contention manager can force the TM implementation to abort or delay some transactions. In this work we consider a contention manager as an integral part of a TM implementation. That is, all the results of the paper apply to the entire TM, including the contention manager.

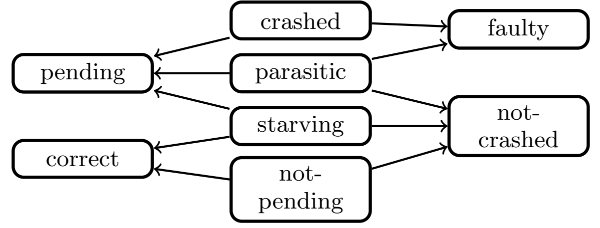

Figure 2: Classes of processes. An arrow from class c_1 to class c_2 means that every process which belongs to c_1 also belongs to c_2.

The order in which processes invoke events is determined by a *scheduler*. Processes and TM implementations have no control over a scheduler. The scheduler decides which process is allowed to send an invocation to the TM implementation at given point in time. These decisions form a *schedule* which is a finite or an infinite sequence of process identifiers.

2.3 Process Failures

We say that process p_k is *pending* in infinite history H if H has only a finite number of commit events C^k. Process p_k *crashes* in infinite history H if $H|p_k \in \Sigma_k^*$. That is, from some point in time p_k stops sending invocation events.

Intuitively, a *parasitic* process is a process that keeps executing operations but, from some point in time, never attempts to commit (by invoking operation $tryC$) when given a chance to do so. Note that if starting from some moment in time every transaction of a process is prematurely aborted, i.e. aborted before the process invokes a commit request, we cannot tell whether the process will invoke a commit requests, if it given a chance to do so. Therefore, we consider such processes as correct. Consider any infinite history H, and process p_k in H. If process p_k from some point in time executes infinitely many operations without being aborted and without attempting to commit, then p_k is parasitic. On the contrary, if p_k invokes operation $tryC^k$ or is aborted infinitely many times, then p_k is not parasitic. Formally, we say that process p_k is parasitic in infinite history H if $H|p_k$ is infinite and in history $H|p_k$ there are only a finite number of invocations $tryC^k$ and abort events A^k. If a process does not crash, is not parasitic, and is pending in infinite history H, then it is *starving* in H.

We say that process p_k is *correct* in infinite history H if p_k is not parasitic in H and does not crash in H. If a process is not correct in H, then it is *faulty* in H. Figure 2 depicts the relations between different classes of processes.

We define a *crash-prone system* (*parasitic-prone system*) Sys to be a system in which at least one process can crash (be parasitic). A *fault-prone system* Sys is a system which is crash-prone or parasitic-prone.

2.4 Safety properties of TM

A safety property S states that some events should never happen. Intuitively a safety property of TM implementations should capture the fact that all events within a transaction appear to other transactions as if they occur instanta-

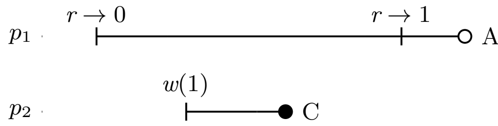

Figure 3: A history which is not opaque but strictly serializable. Hereafter, for simplicity, process and t-variable identifiers in operations are omitted, $r \to v$ means that a process invokes a read operation and receives value v, $w(v)$ means that a process invokes a write operation to write value v and receives *ok*, C means that a process invokes a commit request and receives a commit event, A means that a process invokes a commit request and receives an abort event.

neously. If a transaction is committed, then all the changes made by write operations within the transaction are made visible to other transactions; otherwise all the changes are rolled back. We consider two safety properties of TM implementations: strict serializability S_s and opacity S_o. Intuitively, strict serializability requires every committed transaction to observe a consistent state of the system [19], while opacity requires every transaction (even aborted or unfinished) to observe a consistent state of the system [9].

We say that history H is *equivalent* to history H' if for every process $p_k \in P$ we have $H|p_k = H'|p_k$. We obtain the *completion* $comp(H)$ of finite history H by aborting every transaction which is neither committed nor aborted, i.e. by adding to the end of the history corresponding abort events. If $comp(H) = H$, then H is a *complete* history. We say that finite history H' *preserves the real time order* of finite history H if for any two transactions T_1 and T_2 in H if $T_1 <_H T_2$, then $T_1 <_{H'} T_2$. Let H_s be a complete sequential history and T_j be a transaction in H. Denote by $visible(T_j)$ the longest subsequence of H_s such that for every transaction T_i in the subsequence, either $j = i$ or $T_i <_{H_s} T_j$. Transaction T_j is *legal* in H_s if for every t-variable $x \in X$ history $visible(T_j)$ respects the sequential specification of x, i.e. for every transaction T_i in $visible(T_j)$ and every response event v^k in T_i, v is the value of the previous write to x invocation event in $visible(T_j)$ or v is the initial value of x if there are no write to x invocation events in $visible(T_j)$ before v^k.

A finite history H is *opaque* if there exists a sequential history H_s equivalent to $comp(H)$, such that H_s preserves the real-time order of $comp(H)$, and every transaction in H_s is legal. A finite history H is *strictly serializable* if there exists a sequential history H_s equivalent to H', where H' is obtained from H by removing every aborted or unfinished transaction, such that H_s preserves the real-time order of H, and every transaction in H_s is legal. Let M be a TM implementation represented by I/O automaton F. We say that M ensures opacity (strict serializability) iff every finite history H of F is opaque (strictly serializable).

For example, the history in Figure 1 is opaque, while the history in Figure 3 is not opaque but strictly serializable.

3. LIVENESS OF A TM

We introduce in this section the concept of a *TM-liveness* property and we give examples of such properties.

3.1 TM-liveness Properties

Basically, a TM-liveness property states whether some process p_k should make progress in some infinite history H. Clearly, progress cannot be required for crashed or parasitic processes: these processes have executions with a finite number of *tryC* operation invocations. We define a TM-liveness property as a weakening of the strongest TM-liveness property. The strongest TM-liveness property guarantees that in every infinite history of a TM implementation every correct process makes progress.

Formally, a correct process p_k in infinite history H *makes progress* in H iff p_k is not pending H. Let H_{TM} be the set of all infinite well-formed histories.

We define *local progress*, which is analogous to wait-freedom in shared memory, as a set L_{local} of histories from H_{TM} such that infinite history $H \in H_{TM}$ belongs to L_{local} iff every correct process in H makes progress in H, or H does not have any correct process. A *TM-liveness* property L is a set of infinite histories such that $L_{local} \subseteq L \subseteq H_{TM}$. Given two TM-liveness properties L_1 and L_2, we say that L_1 is weaker (stronger) than L_2 iff $L_2 \subseteq L_1$ ($L_1 \subseteq L_2$). An infinite history H ensures TM-liveness property L iff $H \in L$.

Intuitively a TM implementation ensures a TM-liveness property iff every infinite history of the implementation ensures the property. However, we have to exclude the case when the implementation cannot produce an infinite history, i.e. when the implementation does not send response events to any invocation event of any process. We say that I/O automaton F that models some TM implementation is *live* iff every finite history H of F can be extended to some infinite history $H \cdot H'$ of F.

Let M be a TM implementation modeled by I/O automaton F. TM implementation M ensures TM-liveness property L iff F is live and every infinite history H of F ensures L.

3.2 Examples of TM-liveness Properties

3.2.1 Local Progress

A TM implementation M ensures local progress if M guarantees that every correct process makes progress.

For example, Figure 4 shows an infinite history which ensures local progress in a system with two processes and one t-variable. Both processes make progress (are not pending) in the history.

As we prove in this paper, implementing a TM that ensures opacity and 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 system that is both crash-free and parasitic-free is possible. It suffices to use a simple TM that synchronizes all transactions using a single global starvation-free lock, and thus never aborts any transaction.

3.2.2 Global Progress

A TM implementation M ensures *global progress* if M guarantees that some correct process makes progress. We define global progress, as a TM-liveness property L_{global} such that infinite history $H \in H_{TM}$ belongs to L_{global} iff at least one correct process in H makes progress in H, or H does not have correct processes.

Figure 5 depicts an infinite history which ensures global progress in a system two processes and one t-variable. Both

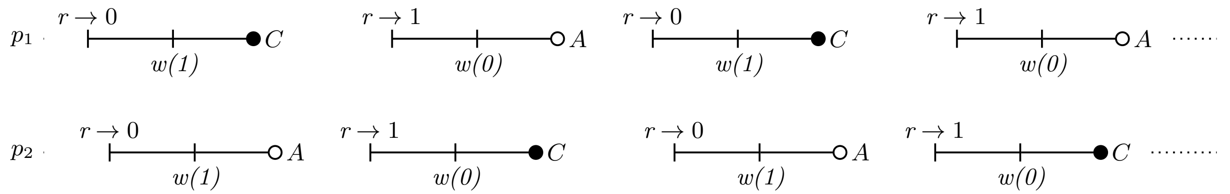

Figure 4: An infinite history with two processes and one t-variable. Each process executes an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0).

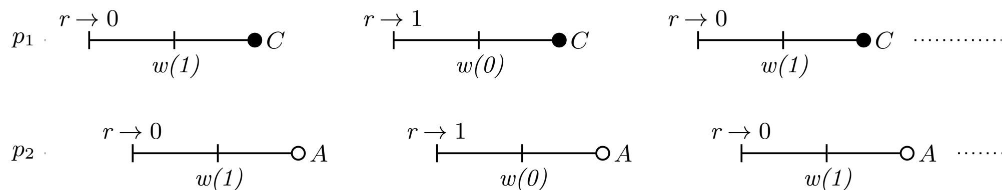

Figure 5: An infinite history with two processes and one t-variable. Processes execute an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0).

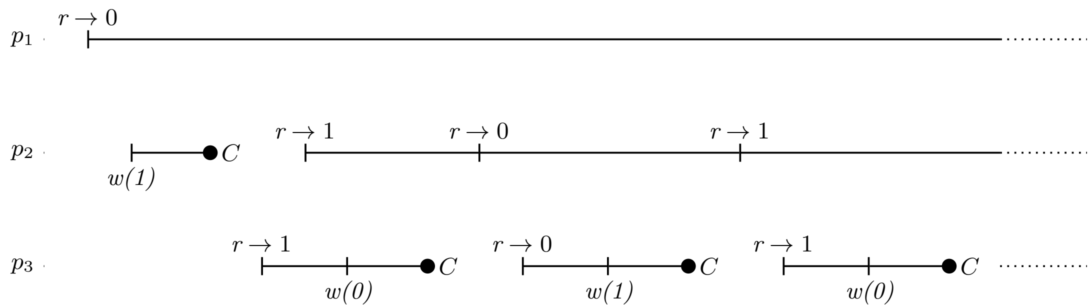

Figure 6: An infinite history with three processes and one t-variable. Process p_1 starts a transaction by invoking a read operations, but then it crashes. Process p_2 executes two transactions, but it becomes parasitic in the second transaction. Process p_3 executes an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0).

of the processes are correct in the history. However, only process p_1 makes progress in the history.

3.2.3 Solo Progress

A TM implementation M ensures *solo progress* if M guarantees that every correct process which eventually runs alone makes progress. A correct process *runs alone* if starting from some point in time it is concurrent only to processes which are faulty. Note that in the TM context the definition of a process running alone is different from the definition in classical shared-memory systems: in the TM context a process p_k runs alone even when other process concurrently invoke operations, but p_k is the only one which invokes infinitely many commit requests.

Formally, a process p_k runs alone in infinite history H iff p_k is correct in H and no other process is correct in H. We define solo progress, as a TM-liveness property L_{solo} such that infinite history $H \in H_{TM}$ belongs to L_{solo} iff a process that runs alone in H makes progress in H, or H does not have a process that runs alone in H.

Figure 6 depicts an infinite history H_{solo} which ensures solo progress in a system with three processes and one t-variable. Process p_1 crashes, p_2 is parasitic, and p_3 runs alone and makes progress (is not pending).

Obstruction-free TM implementations [9, 12] ensure solo progress in parasitic-free systems. Lock-based TM implementations, such as TinySTM [6] and SwissTM [5], ensure solo progress in systems that are both parasitic-free and crash-free. lock-based TMs that use lazy acquire, however, such as TL2 [3], ensure solo progress in crash-free systems.

4. IMPOSSIBILITY OF LOCAL PROGRESS

Like in any distributed problem, each history of a TM implementation can be thought of as a game between the environment and the implementation. The *environment* consisting of processes and a scheduler decides on inputs (operation invocations) given to the implementation and the implementation decides on outputs (responses) returned to the environment. To prove that there is no TM implementation that ensures both opacity and local progress in a fault prone system we use the environment as an adversary that acts against the implementation. The environment wins if the resulting infinite history violates local progress. To prove the impossibility result, we show a wining strategy for the environment.

THEOREM 1. *For every fault-prone system, there does not exist a TM implementation that ensures both local progress and opacity in that system.*

PROOF. Assume otherwise, i.e. that there exists a fault-prone system *Sys* for which there exists a TM implementation M modeled by I/O automaton F that ensures local progress and opacity in *Sys*. To find a contradiction, we exhibit a winning strategy (Strategies 1 and 2 below) for the environment resulting in an infinite history of F which does not ensure local progress.

For simplicity we prove the result for TM implementations that support obstruction-free read and write operations. However, the result holds when the individual operations are not obstruction-free: obstruction freedom ensures that the implementation can produce an infinite history which corresponds to an execution of Strategy 1. If it cannot produce an infinite history, then the implementation is not live and thus does not ensure local progress. Moreover, the result holds for more powerful shared objects that can implement objects supporting read and write operations.

By definition, fault-prone system *Sys* is a system in which at least one process can crash or be parasitic. We thus consider two different cases:

Sys is crash-prone.

Consider two processes p_1 and p_2 and the environment that interacts with M using the following strategy:

Strategy 1.

1. **Step 1.** Process p_1 invokes a read operation on t-variable x and receives as a response v'^1 or A^1. The strategy goes to Step 2.

2. **Step 2.** Process p_2 invokes a read operation on t-variable x and receives as a response v''^2 or A^2. If the response is A^2, then the strategy repeats Step 2. Otherwise p_2 invokes an operation on x, which writes to x (I) value $v'+1$, if p_1 received v'^1 in Step1, or (II) value $v''+1$, if p_1 received A^1 in Step1, and receives as a response ok^2 or A^2. If the response is A^2, then the strategy repeats Step 2. Otherwise p_2 invokes $tryC^2$ operation and receives a response C^2 or A^2. If the response is A^2, the strategy repeats Step 2. Otherwise the strategy goes to Step 3.

3. **Step 3.** If p_1 received A^1 in Step 1, then the strategy goes to Step 1. Otherwise process p_1 invokes a write operation on t-variable x which writes value $v''+1$ to x, and then receives a response. If the response is A^1, then the strategy goes to Step 1. Otherwise p_1 invokes $tryC^1$ operation and receives a response. If the response is A^1, the strategy goes to Step 1. Otherwise the strategy stops.

We first show that there exists an infinite history of F corresponding to an execution of Strategy 1. To do so, we prove that Strategy 1 never terminates. Since individual operations of the implementation are obstruction-free, then the strategy terminates iff at Step 3 process p_1 is returned C^1 by F.

Assume some finite history H_f of F corresponding to an execution of Strategy 1 such that the last event in H_f is C^1. Since M ensures opacity, there exists a sequential finite history H_s which is equivalent to $comp(H_f)$, preserves the real-time order of $comp(H_f)$, and every transaction in H_s is legal. Since history H_f has no transactions which are neither committed nor aborted, then $comp(H_f) = H_f$. Hence H_s is equivalent to H_f and preserves the real-time order of H_f. Since H_s is a sequential history and preserves the real-time order of H_f, then H_s could only have one of the following forms, where H'_s is a prefix of H_s:

1. $H_s = H'_s \cdot x.read^1() \cdot v'^1 \cdot x.write^1(v''+1) \cdot ok^1 \cdot tryC^1 \cdot C^1 \cdot x.read^2() \cdot v''^2 \cdot x.write^2(v'+1) \cdot ok^2 \cdot tryC^2 \cdot C^2$

2. $H_s = H'_s \cdot x.read^2() \cdot v''^2 \cdot x.write^2(v'+1) \cdot ok^2 \cdot tryC^2 \cdot C^2 \cdot x.read^1() \cdot v'^1 \cdot x.write^1(v''+1) \cdot ok^1 \cdot tryC^1 \cdot C^1$.

In the first case, the last transaction executed by process p_2 is not legal in H_s, because p_2 reads value v'' from t-variable x the value of which is $v''+1$ and this violates the semantics of x. In the second case, the last transaction executed by process p_1 is not legal in H_s, because p_1 reads value v' from t-variable x the value of which is $v'+1$, this leads to violation

of the specification of x. Thus, H_f is not opaque. Since every history H_f of F that ends with commit event C^1 is not opaque and M ensures opacity, then H_f is not a history of F corresponding to the execution of the strategy. In other words, every history of F corresponding to the execution of Strategy 1 is infinite.

Consider some infinite history H of F corresponding to the execution of the above strategy. Since process p_1 never receives commit event C^1 from M, then p_1 is pending in H. Since *Sys* is crash-prone, then process p_1 can crash in history H. Therefore, we focus on the following two cases:

- **Process p_1 crashes in history H.** According to the strategy, process p_1 crashes in infinite history H iff process p_2 is pending and invokes infinitely many operations. Process p_2 invokes infinitely many operations iff the strategy executes infinitely many iterations of Step 2. At each iteration of Step 2 process p_2 either receives abort event A^2 or invokes operation $tryC^2$, thus p_2 is correct in H. Since M ensures local progress and p_2 is correct in H, then process p_2 is not pending: a contradiction. Thus, H does not ensure local progress.

- **Process p_1 does not crash in history H.** Since H is infinite and p_1 does not crash in H, then according to the strategy p_1 invokes infinitely many operations and receives infinitely many abort events. Thus, p_1 is a correct process in H. Since M ensures local progress, then p_1 makes progress in H, which means that eventually p_1 is returned commit event $C^1$and history H is not infinite: a contradiction. Thus, H does not ensure local progress.

Sys is parasitic-prone. Consider two processes p_1 and p_2 and the environment that interacts with M using the following strategy:
Strategy 2.

1. **Step 1.** Process p_1 invokes a read operation on t-variable x and receives as a response v'^1 or A^1. Otherwise process p_2 invokes a read operation on x and receives as a response v''^2 or A^2. If the response is A^2, then the strategy repeats Step 1. Otherwise p_2 invokes a write operation which writes to x (I) value $v' + 1$, if p_1 received v'^1, or (II) value $v'' + 1$, if p_1 received A^1, and then p_2 receives a response. If the response is A^2, then the strategy repeats Step 1. Otherwise p_2 invokes $tryC^2$ operation and receives a response. If the response is A^2, then the strategy repeats Step 1. Otherwise the strategy goes to Step 2.

2. **Step 2.** If p_1 received A^1 in Step 1, then the strategy goes to Step 1. Process p_1 invokes a write operation on x which writes value $v'' + 1$ to x, and then p_1 receives a response. If the response is A^1, then the strategy goes to Step 1. Otherwise p_1 invokes $tryC^1$ operation and receives a response. If the response is A^1, then the strategy goes to Step 1. Otherwise the strategy stops.

First, we prove that Strategy 2 never terminates, i.e. that at Step 2 process p_1 is never returned C^1 by M in any history of M corresponding to an execution of the strategy. Assume some finite history H_f of F corresponding to an execution of Strategy 2 such that the last event in H_f is C^1. Since M ensures opacity, there exists a sequential finite history H_s which is equivalent to $comp(H_f)$, preserves the real-time order of $comp(H_f)$, and every transaction in H_s is legal. Since history H_f has no transaction which are neither committed nor aborted, then $comp(H_f) = H_f$. Hence H_s is equivalent to H_f and preserves the real-time order of H_f. Since H_s is a sequential history and preserves the real-time order of H_f, then H_s could only have one of the following forms, where H'_s is a prefix of H_s:

1. $H_s = H'_s \cdot x.read^1() \cdot v'^1 \cdot x.write^1(v'' + 1) \cdot ok^1 \cdot tryC^1 \cdot C^1 \cdot x.read^2() \cdot v''^2 \cdot x.write^2(v' + 1) \cdot ok^2 \cdot tryC^2 \cdot C^2$

2. $H_s = H'_s \cdot x.read^2() \cdot v''^2 \cdot x.write^2(v' + 1) \cdot ok^2 \cdot tryC^2 \cdot C^2 \cdot x.read^1() \cdot v'^1 \cdot x.write^1(v'' + 1) \cdot ok^1 \cdot tryC^1 \cdot C^1$.

In the first case, the last transaction executed by process p_2 is not legal in H_s, because p_2 reads value v'' from t-variable x the value of which is $v'' + 1$ and this violates the semantics of x. In the second case, the last transaction executed by process p_1 is not legal in H_s, because p_1 reads value v' from t-variable x the value of which is $v' + 1$, this leads to violation of the specification of x. Thus, H_f is not opaque. Since every history H_f of F that ends with commit event C^1 is not opaque and M ensures opacity, then H_f is not a history of F corresponding to the execution of the strategy. In other words, every history of F corresponding to the execution of Strategy 2 is infinite.

Consider now some infinite history H of F corresponding to the execution of the above strategy. Since process p_1 never receives commit event C^1 from M, then p_1 is pending in H. Since *S* is parasitic-prone, then process p_1 can be parasitic in history H. Therefore, we focus on the following two cases:

- **Process p_1 is parasitic in history H.** According to the strategy, process p_1 is parasitic in infinite history H iff process p_2 is pending and invokes infinitely many operations at Step 1 without receiving a commit event C^2. Process p_2 invokes infinitely many operations iff the strategy executes infinitely many iterations of Step 1. At each iteration of Step 1 process p_2 either receives abort event A^2 or invokes operation $tryC^2$, thus p_2 is correct in H. Since M ensures local progress, then p_2 makes progress in H, i.e. process p_2 is not pending: a contradiction. Thus, H does not ensure local progress.

- **Process p_1 is not parasitic in history H.** According to Strategy 2 H is infinite iff process p_1 invokes infinitely many operations. Since p_1 invokes infinitely many operations and p_1 is pending in H, then p_1 receives infinitely many abort events in H. Thus, p_1 is correct in H. Since M ensures local progress, then p_1 makes progress in H, which means that eventually p_1 is returned commit event C^k and H is finite: a contradiction. Thus, H does not ensure local progress.

□

5. GENERALIZING THE IMPOSSIBILITY

We generalize here the result of the previous section; namely, we determine a larger class of TM-liveness properties that are impossible to implement together with strict serializability, which is weaker than opacity, in a fault-prone system.

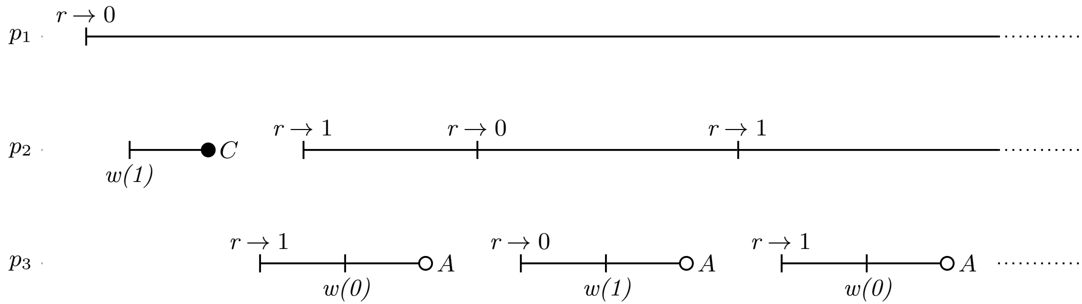

Figure 7: An infinite history with three processes and one t-variable. Process p_1 starts a transaction by invoking a read operations, but then it crashes. Process p_2 executes two transactions, but it becomes parasitic in the second transaction. Process p_3 executes an infinite number of aborting transactions which read value 0 (read value 1) and write value 1 (write value 0).

5.1 Classes of TM-liveness properties

Nonblocking TM-liveness properties.

Intuitively, we say that a TM-liveness property is *nonblocking* if it guarantees progress for every correct process that eventually runs alone. More precisely, a TM-liveness property L is nonblocking iff for every $H \in L$ if some process runs alone in H, then the process makes progress in H.

For example, Figure 4, Figure 5, and Figure 6 show infinite histories which ensure nonblocking TM-liveness properties while Figure 7 shows an infinite history which does not ensure any nonblocking TM-liveness property. TM-liveness properties that are not nonblocking are called *blocking*. Local progress, global progress, and solo progress are nonblocking. Note that solo progress is the weakest among nonblocking properties while local progress is the strongest among nonblocking properties.

Biprogressing TM-liveness properties.

Intuitively, we say that a TM-liveness property L is a *biprogressing* property if for every infinite history it guarantees that at least two correct processes make progress. More precisely, a TM-liveness property $L = \{L^1, \dots, L^n\}$ is *biprogressing* iff for every $H \in L$ if at least two processes are correct in H, then at least two processes make progress in H.

For example, Figure 4 and Figure 6 show infinite histories which ensure a biprogressing property while Figure 5 shows an infinite history which does not ensure any biprogressing property. Local progress is a biprogressing property while global progress and solo progress are not biprogressing.

5.2 Generalized Result

We show that TM-liveness properties that are nonblocking and biprogressing are impossible to implement together with strict serializability in any fault-prone system. We start by stating the following lemma, which says, intuitively, that there exists a history in which a process executing infinitely many transactions can block the progress of all other processes if the TM ensures any nonblocking TM-liveness property. The proof of the lemma follows the same line of reasoning as in Theorem 1.

LEMMA 1. *For every TM implementation that ensures strict serializability and a nonblocking TM-liveness property in any fault-prone system, there exists an infinite history H of the implementation such that at least two processes are correct in H and at most one process makes progress in H.*

PROOF. Let M be a TM implementation ensuring strict serializability and a nonblocking TM-liveness property in a fault-prone system *Sys* and F be its I/O automaton representation. To exhibit a history in which at least two processes are correct and at most one process makes progress we consider a game between the environment and the implementation. The environment acts against the implementation and wins the game if the resulting history satisfies the requirements of the lemma.

By definition, fault-prone system *Sys* is a system in which at least one process can crash or be parasitic. We thus consider two different cases:

Sys is crash-prone. Consider two processes p_1 and p_2 that interact with M. The environment uses Strategy 1 to win the game. We can show that Strategy 1 never terminates using the same line of reasoning as in Theorem 1.

Consider some infinite history H corresponding to an execution of the strategy. Since *Sys* is crash-prone, process p_1 either crashes in history H or does not crash in H.

Assume that process p_1 crashes in history H. According to the strategy, process p_1 can crash in infinite history H only if process p_2 is pending and invokes infinitely many operations, i.e. only if p_2 is returned an infinite number of abort events at Step 2. Since p_2 is returned an infinite number of abort events, p_2 is correct in H . Because after some time only process p_2 executes operations in H (i.e. p_2 runs alone in H) and M ensures a TM-liveness property which is nonblocking, then p_2 makes progress in H, i.e. process p_2 is not pending: a contradiction. Thus, p_1 cannot crash in H.

According to the strategy, p_2 cannot crash in H since Step 2 is repeated infinitely often. Since Step 2 and Step 1 are repeated infinitely often (because p_1 does not crash in H), then p_2 receives infinitely many commit events C^2, i.e. p_2 is correct. Since process p_1 is returned infinitely many abort events A^1 at Step 1 or Step 3, process p_1 is correct. Thus, in history H both of the processes are correct and at most one process makes progress (since p_1 is never returned C^1).

Sys is parasitic-prone. Consider two processes p_1 and p_2 that interact with M. The environment uses Strategy 2 to

win the game. We can show that Strategy 2 never terminates using the same line of reasoning as in Theorem 1.

Consider some infinite history H corresponding to an execution of the strategy. Since *Sys* is parasitic-prone, process p_1 is either parasitic or not in H.

Assume that p_1 is parasitic in H. According to the strategy, p_1 can be parasitic only if p_2 is pending in H and returned A^2 infinitely often (i.e. correct). Since a correct process p_2 runs alone in H and M ensures a nonblocking TM-liveness property, then p_2 makes progress in H: a contradiction. Thus, p_1 cannot be parasitic in H.

According to the strategy, processes p_1 and p_2 do not crash because both of the processes invoke infinitely many read requests. Process p_2 cannot be parasitic in H since p_2 either invokes $tryC^2$ or is returned A^2 infinitely often at Step 1. Thus, in history H both of the processes are correct and at most one process makes progress (since p_1 is never returned C^1).

Sys is not crash-free or parasitic free. Since in *Sys* any number of processes can crash or be parasitic there are no restrictions on a strategy used by the environment. Thus, we can use Strategy 1 (or Strategy 2) to exhibit an infinite history that does not ensure local progress. □

By definition, a biprogressing TM-liveness property should ensure progress for at least two correct processes in every infinite history. While, by the above lemma, if the property is also nonblocking, then we can find an infinite history of any TM implementation in a fault prone system in which at least two processes are correct and at most one process makes progress: a contradiction. Thus, we have the following theorem.

THEOREM 2. *For every fault-prone system and every TM-liveness property L which is nonblocking and biprogressing there is no TM implementation that ensures strict serializability and L in that system.*

6. CONCLUDING REMARKS

We propose a framework to formally reason about liveness properties of TMs and introduce the very notion of a TM-liveness property. We prove in particular that in a system with faulty processes (crashes or parasitic), local progress cannot be ensured together with opacity, the safety property typically ensured by most TMs. We presented this impossibility result in its direct and then general form.

Local progress of transactional memory implementations is analogous to wait-freedom in concurrent computing which is the ultimate classical liveness property (for non-transactional objects) in concurrent computing. Just like wait-freedom makes sure processes do not wait for each other, local progress ensures that transactions of different processes do not wait for each other. The fact that wait-freedom was shown to be possible to implement led researchers to focus on how to achieve it efficiently. The fact that local progress is impossible to implement means that researchers have to find alternatives.

There are several ways to circumvent our impossibility result. One way is to weaken safety or TM-liveness property requirements, for example, to require only global progress. There are implementations that ensure opacity and global progress, e.g., OSTM [7]. A second way is to assume that all transactions are static and predefined. That is a TM knows exactly which operations, on which shared variables, will be invoked in a transaction. In that case transactions can be viewed as simple operations and one can apply classical universal construction to ensure local progress [11]. However, assuming static transactions may be too limiting for certain applications. A third way is to assume a different system model instead of the multi-threaded programming model. For example, [21] shows a TM implementation that ensures local progress in an asynchronous multicore system model which assumes that a transaction can be executed by different processes and that some process crashes are detectable by the runtime system.

As we pointed out, this paper is a first step towards understanding the liveness of TMs and many problems are open. It would be interesting to determine precisely the strongest liveness property that can be ensured by a TM as well as study the impact on the impossibility of reducing the number of possible faults that a TM can face. Another possible direction for future work would be to generalize the impossibility result even further by considering classes of TM-liveness properties that guarantee progress for processes with higher priority.

7. REFERENCES

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, jan 2011.

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

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of DISC'06*, pages 194–208. Springer-Verlag, 2006.

[4] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259:245–261, dec 2009.

[5] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of ACM PLDI'09*, pages 155–165. ACM, 2009.

[6] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of ACM PPoPP'08*, pages 237–246. ACM, 2008.

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

[8] R. Gawlick, R. Segala, J. F. Søgaard-Andersen, and N. A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, mar 1998.

[9] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan and Claypool, 2010.

[10] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan and Claypool, 2010.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, jan 1991.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of ACM PODC'03*, pages 92–101. ACM, 2003.

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

[14] M. Herlihy and N. Shavit. On the nature of progress. In *Proceedings of the 15th international conference on Principles of Distributed Systems*, pages 313–328. Springer-Verlag, 2011.

[15] D. Imbs, J. R. de Mendivil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *Proceedings of ACM PODC'09*, pages 280–281. ACM, 2009.

[16] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, aug 2005.

[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 *Proceedings of ACM SPAA'08*, pages 314–325. ACM, 2008.

[18] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proceedings of ACM POPL'08*, pages 51–62. ACM, 2008.

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

[20] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of ACM PODC'95*, pages 204–213, New York, NY, USA, 1995. ACM.

[21] J.-T. Wamhoff and C. Fetzer. The universal transactional memory construction. In *Proceedings of TRANSACT'11*, 2011.