The Semantics of Progress in Lock-Based Transactional Memory*

Rachid Guerraoui   Michał Kapałka
School of Computer and Communication Sciences, EPFL,
Lausanne, Switzerland

July 15, 2008 (revised: October, 2008)

Abstract

Transactional memory (TM) is a promising paradigm for concurrent programming. Whereas the number of TM implementations is growing, however, little research has been conducted to precisely define TM semantics, especially their progress guarantees. This paper is the first to formally define the progress semantics of lock-based TMs, which are considered the most effective in practice.

We use our semantics to reduce the problems of reasoning about the correctness and computability power of lock-based TMs to those of simple try-lock objects. More specifically, we prove that checking the progress of any set of transactions accessing an arbitrarily large set of shared variables can be reduced to verifying a simple property of each individual (logical) try-lock used by those transactions. We use this theorem to determine the correctness of state-of-the-art lock-based TMs and highlight various configuration ambiguities. We also prove that lock-based TMs have consensus number 2. This means that, on the one hand, a lock-based TM cannot be implemented using only read-write memory, but, on the other hand, it does not need very powerful instructions such as the commonly used compare-and-swap.

We finally use our semantics to formally capture an inherent trade-off in the performance of lock-based TM implementations. Namely, we show that the space complexity of every lock-based software TM implementation that uses invisible reads is at least exponential in the number of objects accessible to transactions.

Keywords. Transactional memory, lock, try-lock, consensus number, impossibility, lower bound, reduction, semantics, verification

1 Introduction

Multi-core processors are predicted to be common in home computers, laptops, and maybe even smoke detectors. To exploit the power of modern hardware, applications will need to become increasingly parallel. However, writing scalable concurrent programs is hard and error-prone with traditional locking techniques. On the one hand, coarse-grained locking throttles parallelism and causes lock contention. On the other hand, fine-grained locking is usually an engineering challenge, and as such is not suitable for use by the masses of programmers.

Transactional memory (TM) [14] is a promising technique to facilitate concurrent programming while delivering comparable performance to fine-grained locking implementations. In short, a TM allows concurrent threads of an application to communicate by executing lightweight, in-memory *transactions*. A transaction accesses shared data and then either commits or aborts. If it commits, its operations are applied to the shared state *atomically*. If it aborts, however, its changes to the shared data are lost and never visible to other transactions.

While a large number of TM implementations have been proposed so far, there is still no precise and complete description of the *semantics* of a TM. Indeed, a correctness criterion for TM, called *opacity*, has been proposed [10], and the progress properties of *obstruction-free* TM implementations have been defined [9]. However, opacity is only concerned with safety—it does not specify when transactions need to commit. (For example, a TM that aborts every transaction could trivially ensure opacity.) Moreover, TM implementations that are considered effective [5], e.g., TL2 [4], TinySTM [7], a version of RSTM [20], BartokSTM [12], or McRT-STM [2] are not obstruction-free. They internally use locking, in order to reduce the overheads of TM mechanisms, and do not ensure obstruction-freedom, which inherently precludes the use of locks.

Lock-based TMs do ensure some progress for transactions, for otherwise nobody would use them. However, this has never been precisely defined. The lack of such a definition hampers the portability of applications that use lock-based TMs, and makes it difficult to reason formally about their correctness or to establish whether any performance limitation is inherent or simply an artifact of a specific implementation.

This paper defines the progress semantics of lock-based TMs. We do so by introducing a new property, which we call *strong progressiveness*,[1] and which stipulates the two following requirements.

*EPFL Technical Report. Submitted for publication.

[1] We call it "strong" by opposition to a weaker form of progressiveness that we also introduce in this paper.

1. A transaction that encounters no *conflict* must be able to commit. (Basically, a conflict occurs when two or more concurrent transactions access the same transactional variable and at least one of those accesses is not read-only.)

2. If a number of transactions have only a "simple" conflict, i.e., on a single transactional variable, then at least one of them must be able to commit.

The former property captures the common intuition about the progress of any TM (see [25]). The second property ensures that conflicts that are easy to resolve do not cause all conflicting transactions to be aborted. This is especially important when non-transactional accesses to shared variables are encapsulated inside unit transactions to ensure strong atomicity [3]. Strong progressiveness, together with opacity and operation-level wait-freedom,[2] is ensured by state of the art lock-based implementations, such as TL2, TinySTM, RSTM, BartokSTM, and McRT-STM.[3]

We use our strong progressive semantics to reduce the problems of reasoning about the correctness and computability power of lock-based TMs to those of simple *try-lock* objects [26, 17]. We first show that proving strong progressiveness of a set of transactions accessing any number of shared variables can be reduced to proving a simple property of every individual logical try-lock that protects those variables. Basically, we prove that if it is possible to say which parts of a TM algorithm can be viewed as logical try-locks (in a precise sense we define in the paper), and if those logical try-locks are *strong*, then the TM is strongly progressive. Intuitively, a try-lock is strong if it guarantees that among processes that compete for the unlocked try-lock, one always acquires the try-lock (most try-locks in the literature that are implemented from compare-and-swap or test-and-set are strong). We illustrate our reduction approach on state-of-the-art lock-based TMs. We formally establish and prove their correctness while highlighting some of their configurations that, maybe unexpectedly, violate the progress semantics.

Then, still using the try-lock reduction, we show that a lock-based TM has *consensus number 2* in the parlance of [13]. The consensus number is a commonly used metric for the computational power of a shared-memory abstraction, and is expressed as the maximum number of processes that can solve a non-trivial agreement problem (namely consensus [13]) in a wait-free manner using this abstraction. The fact that a lock-based TM has consensus number 2 means that such a TM cannot be implemented using only read-write memory instructions, but, on the other hand, powerful instructions such as compare-and-swap are not necessary to implement a lock-based TM.

In fact, we give an implementation of a lock-based TM using read-write and test-and-set instructions. This implementation might be interesting in its own right when compare-and-swap instructions are not available or simply too expensive. Interestingly, we highlight an alternative semantics we call *weak progressiveness* which enables TMs with consensus number 1 and can thus be implemented using only read-write memory. Intuitively, weak progressiveness requires only that a transaction that encounters no conflicts commits. This might be considered a viable alternative to strong progressiveness for "lightweight" lock-based implementations.

We finally use our progress semantics to determine an inherent trade-off between the required memory and the latency of reads in lock-based TMs. This trade-off impacts the performance and/or progress guarantees of a TM but it was never formally established, precisely because of the lack of any precise semantics. We show that the space complexity of every lock-based TM that uses the *invisible reads* strategy[4] is at least exponential in the number of variables available to transactions. This might seem surprising, since it is not obvious that modern lock-based TMs have non-linear space complexity. The exponential (or, in fact, unbounded) complexity comes from the use of timestamps that determine version numbers of shared variables. TM implementations usually reserve a constant-size word for each version number (which gives linear space complexity). However, an overflow can happen and has to be handled in order to guarantee correctness (opacity). As we explain in Section 6.3, this requires (a) limiting the progress of transactions when overflow occurs and (b) preventing read-only transactions from being completely invisible. Concretely speaking, our result means that efficient TM implementations (the ones that use invisible reads) must either intermittently (albeit very rarely) violate progress guarantees, or use unbounded timestamps.

Summary of contributions. To summarize, this paper contributes to the understanding of TM design and implementations by presenting the first precise semantics of a large class of popular TMs—lock-based ones. We precisely define the progress semantics of such TMs and propose reduction approaches to simplify their verification and computational study. We also use our semantics to study their inherent performance bottlenecks.

Roadmap. The rest of the paper proceeds as follows. First, in Section 2, we describe the basic model and terminology used to state our semantics and prove our results. Then, in Section 3, we define the progress semantics of

[2]Wait-freedom [13] requires threads executing operations on transactional data within transactions to make progress independently, i.e., without waiting for each other. Maybe surprisingly, this property can easily be ensured by lock-based TMs.

[3]The source code of the implementations of BartokSTM and McRT-STM is not publicly available. We could thus verify strong progressiveness of those TMs only from their algorithm descriptions in [12] and [2], respectively.

[4]With invisible reads, the reading of transactional variables is performed optimistically, without any (shared or exclusive) locking or updates to shared state. Invisible reads are used by most TM implementations and considered crucial for good performance in read-dominated workloads.

lock-based TMs. In Section 4, we show how to simplify the verification of strong progressiveness. Next, in Sections 5 and 6, we establish the fundamental power and limitations of lock-based TMs. We also discuss in those sections the impact of weakening progress properties. Finally, in Section 7, we discuss possible extensions of the results presented in this paper.

Related work. It is worth noting that there has been an attempt to describe the overall semantics of TMs [25] (including lock-based ones). However, the approach taken there is very low-level—the properties are defined with respect to specific TM protocols and strategies. Our approach is more general: we define semantics that is implementation-agnostic and that is visible through the public interface of a TM to a user. We also show how this semantics can be verified.

There have also been other attempts to describe the semantics of a TM, e.g., in [27, 16, 1, 22, 21]. Those papers, however, focus on safety, i.e., serializability. In [22] there is a notion of progress, but it refers to deadlock-freedom of the whole system (i.e., making sure at least one thread can execute a step at any given time) rather than progress of individual transactions.

2 Preliminaries

2.1 Shared Objects and their Implementations

We consider an asynchronous shared memory system of n processes (threads) $p_1, \ldots, p_n$ that communicate by executing operations on (shared) objects. (At the hardware level, a shared object is simply a word in shared memory with the instructions supported by a given processor, e.g., read, write, or compare-and-swap.) An example of a very simple shared object is a *register*,[5] which exports only *read* and *write* operations. Operation *read* returns the current state (value) of the register, and $write(v)$ sets the state of the register to value v. Hence, a register provides the basic read-write memory semantics.

Consider a single run of any algorithm. A *history* is a sequence of invocations and responses of operations that were executed by processes on (shared) objects in this run. A history of an object x is a history that contains only operations executed on x. (Note here that we assume that events executed in a given run can be totally ordered by their execution time; events that are issued at the same time, e.g., on multi-processor systems, can be ordered arbitrarily.)

An object x may be implemented either directly in hardware, or from other, possibly more primitive, objects, which we call *base objects*. If I_x is an implementation of an object x, then an *implementation* history of I_x is a sequence of (1) invocations and responses of operations on x, and (2) corresponding operations on base objects (called *steps*) that were executed by I_x (i.e., by processes executing I_x) in some run. Hence, intuitively, a history of an object x represents what happened in some run at the (public) interface of x. An implementation history, in addition, shows what steps the implementation of x executed in response to the operations invoked on x.

In algorithms, for simplicity, we assume that base objects such as registers and test-and-set objects are atomic, i.e., linearizable [15]. That is, operations on those objects appear (to the application) as if they happened instantaneously at some unique point in time between their invocation and response events. (For example, in Java, a "volatile" variable is an atomic register, while an object of class `AtomicInteger` is an atomic object that supports operations such as *get*, *set*, *incrementAndGet*, etc.)

However, assuming a weaker memory model does not impact our results: the progress properties we define do not rely on atomicity, strong try-lock objects are not linearizable, and atomic registers of any size can be implemented out of 1-bit safe (the most primitive) registers [19].

If E is an (implementation) history, then $E|p_i$ denotes the restriction of E to events (including steps) executed by process p_i, and $E|x$ denotes the restriction of E to events on object x and steps of the implementation of x. We assume that processes execute operations on objects sequentially. That is, in every restriction $E|p_i$ of an (implementation) history E, no two operations and no two steps overlap.

We focus on object implementations that are *wait-free* [13]. Intuitively, an implementation I_x of an object x is wait-free if a process that invokes an operation on x is never blocked indefinitely long inside the operation, e.g., waiting for other processes. Hence, processes can make progress independently of each other. More precisely:

Definition 1 *An implementation I_x of an object x is* wait-free, *if whenever any process p_i invokes an operation on x, p_i returns from the operation within a finite number of its own steps.*

2.2 Transactional Memory (TM)

A TM enables processes to communicate by executing transactions. For simplicity, we will say that a transaction T performs some action, meaning that the process executing T performs this action within the transactional context of T. A transaction T may perform operations on *transactional variables*, which we call *t-variables* for short. For simplicity, we assume that every t-variable x supports only two operations: *read* that returns the current state (value) of x, and $write(v)$ that sets the state of x to value v. We discuss in Section 7 what changes when t-variables are arbitrary objects, i.e., objects that have operations beyond *read* and *write* (e.g., *incrementAndGet*). Note, however, that most existing TMs either provide only read-write t-variables (e.g., word-based TMs), or ef-

[5]Note that we use here the term "register" in its distributed computing sense: a read-write abstraction.

fectively treat all operations on t-variables as reads and writes (e.g., without exploiting the commutativity relations between non-read-only operations).

Each transaction has its own unique identifier, e.g., T_1, T_2, etc. A transaction T_k may access (read or write) any number of t-variables. Then, T_k may either *commit* or *abort*. We assume that once T_k commits or aborts T_k does not perform any further actions. In this sense, restarting a transaction T_k (i.e., the computation T_k was supposed to perform) is considered in our model as a different transaction (with a different identifier).

We can treat a TM as an object with the following operations:

- $tread_k(x)$ and $twrite_k(x, v)$ that perform, respectively, a *read* or a $write(v)$ operation on a t-variable x within a transaction T_k,
- $tryC_k$ that is a request to commit transaction T_k,
- $tryA_k$ that is a request to abort transaction T_k.

Each of the above operations can return a special value A_k that indicates that the operation has failed and the respective transaction T_k has been aborted. Operation $tryC_k$ returns value C_k if committing T_k has been successful. Operation $tryA_k$ always returns A_k (i.e., it always succeeds in aborting transaction T_k).

The above operations of a TM, in some form, are either explicitly used by a programmer (e.g., in TL2, TinySTM, RSTM), or inserted by a TM-aware compiler (e.g., in McRT-TM, Bartok-STM). Even if the compiler is responsible for inserting those operations, the programmer must specify which blocks of code are parts of transactions, and retains full control of what operations on which t-variables those transactions perform. Hence, in either case, this TM interface is visible to a programmer, and so are properties defined with respect to this interface.

If H is an (implementation) history of a TM object, then $H|T_k$ denotes the restriction of H to only events of transaction T_k. We say that a transaction T_k is in a history H, and write $T_k \in H$, if $H|T_k$ is a non-empty sequence.

Let H be any history and T_k be any transaction in H. We say that T_k is *committed* in H, if H contains response C_k of operation $tryC_k$. We say that T_k is *aborted* in H, if H contains response A_k of any TM operation.

We say that a transaction T_k *follows* a transaction T_i in a history H, if T_i is committed or aborted in H and the first event of T_k in H follows the last event of T_i in H. If neither T_k follows T_i in H, nor T_i follows T_k in H, then we say that T_i and T_k are *concurrent* in H.

We assume that every transaction itself is sequential. That is, for every history H of a TM and every transaction $T_k \in H$, $H|T_k$ is a sequence of non-overlapping TM operations. Clearly, operations of different transactions can overlap. We also assume that each transaction is executed by a single process, and that each process executes only one transaction at a time (i.e., transactions at the same process are never concurrent).

2.3 Try-Locks

All lock-based TMs we know of use (often implicitly) a special kind of locks, usually called *try-locks* [26]. Intuitively, a try-lock is an object that provides mutual exclusion (like a lock), but does not block processes indefinitely. That is, if a process p_i requests a try-lock L but L is already acquired by a different process, p_i is returned the information that its request failed instead of being blocked waiting until L is released.

Try-locks keep the TM implementation simple and avoid deadlocks. Moreover, if any form of fairness is needed, it is provided at a higher level than at the level of individual locks—then more information about a transaction can be used to resolve conflicts and provide progress. Ensuring safety and progress can be effectively separate tasks.

More precisely, a try-lock is an object with the following operations:

1. *trylock*, that returns *true* or *false*; and
2. *unlock*, that always returns *ok*.

Let L be any try-lock. If a process p_i invokes *trylock* on L and is returned *true*, then we say that p_i has *acquired* L. Once p_i acquires L, we say that (1) p_i *holds* L until p_i *invokes* operation *unlock* on L, and (2) L is *locked* until p_i *returns* from operation *unlock* on L. (Hence, L might be locked even if no process holds L—when some process that was holding L is still executing operation *unlock* on L.)

Every try-lock L guarantees the following property, called *mutual exclusion*: no two processes hold L at the same time.

For simplicity, we assume that try-locks are not reentrant. That is, a process p_i may invoke *trylock* on a try-lock L only when p_i does not hold L. Conversely, p_i may invoke *unlock* on L only when p_i holds L.

Intuitively, we say that a try-lock L is *strong* if whenever several processes compete for L, then one should be able to acquire L. This property corresponds to deadlock-freedom, livelock-freedom, or progress [24] properties of (blocking) locks.

Definition 2 *We say that a try-lock L is* strong, *if L ensures the following property, in every run: if L is not locked at some time t and some process invokes operation trylock on L at t, then some process acquires L after t.*

While there exists a large number of lock implementations, only a few are try-locks or can be converted to try-locks in a straightforward way. The technical problems of transforming a queue (blocking) lock into a try-lock are highlighted in [26]. It is trivial to transform a typical TAS or TATAS lock [24] into a strong try-lock (e.g., Algorithm 4 in Section 5.2).

3 Progress of a Lock-Based TM

Lock-based TMs are TM implementations that use (internally) mutual exclusion to handle some phases of a transaction. Most of them use some variant of the two-phase locking protocol, well-known in the database world [6].

From the user's perspective, however, the choice of the mechanism used internally by a TM implementation is not very important. What is important is the semantics the TM manifests on its public interface, and the time/space complexities of the implementation. If those properties are known, then the designer of a lock-based TM is free to choose the techniques that are best for a given hardware platform, without the fear of breaking existing applications that use a TM.

As we already mentioned, the correctness criterion for TMs, including lock-based ones, is usually *opacity* [10]. This property says, intuitively, that (1) committed transactions should appear as if they were executed sequentially, in an order that agrees with their real-time ordering, (2) no transaction should ever observe the modifications to shared state done by aborted or live transactions, and (3) all transactions, including aborted and live ones, should always observe a consistent state of the system. The first two properties correspond, roughly, to the classical database properties: strict serializability [23] and the strongest variant of recoverability [11], respectively. The last property is unique to TMs, and needs to be ensured to prevent unexpected crashes or incorrect behavior of applications that use a TM.

However, opacity is not enough. A TM that always aborts every transaction, or that blocks transactions infinitely long, could ensure opacity and still be useless from the user's perspective. In this section, we define the *progress* properties of a lock-based TM. These involve individual operations of transactions, where it is typical to require *wait-freedom*, and entire transactions, for which we will require our notion of *strong progressiveness*.

3.1 Liveness of TM Operations

If a process p_i invokes an operation (*tread*, *twrite*, *tryC*, or *tryA*) on a TM, we expect that p_i eventually gets a response from the operation. The response might be the special value A_k that informs p_i that its current transaction T_k has been aborted.

We assume that each implementation of a TM is a wait-free object. That is, a TM ensures wait-freedom on the level of its operations. This property is indeed ensured by many current lock-based TMs: if a transaction T_k encounters a conflict, T_k is immediately aborted and the control is returned to the process executing T_k.

Note that a TM may use a *contention manager* to decide what to do in case of a conflict. A contention manager is a logically external module that can reduce contention by delaying or aborting some of the transactions that conflict. In principle, a contention manager could make transactions wait for each other, in which case wait-freedom would be violated. However, such contention managers change the progress properties of a TM significantly and as such should be considered separately.

Operation wait-freedom may also be violated periodically by some TM mechanisms that handle overflows. While those can be unavoidable, as we discuss in Section 6.3, they are executed very rarely. Moreover, one can easily predict when they could start. In this sense, wait-freedom can be guaranteed except for some short periods that can be signalled in advance to processes by, e.g., setting a global flag.

3.2 Progress of Transactions

Intuitively, a transaction makes progress when it commits. One would like most transactions to commit, except those that were explicitly requested by the application to be aborted (using a *tryA* operation of a TM). However, a TM may be often forced to abort some transactions when the conflicts between them cannot be easily resolved. We will call such transactions *forcefully aborted*. The *strong progressiveness* property we introduce here defines when precisely a transaction can be forcefully aborted.

Intuitively, strong progressiveness says that (1) if a transaction has no *conflict* then it cannot be forcefully aborted, and (2) if a group of transactions conflict on a single t-variable, then not all of those transactions can be forcefully aborted. Roughly speaking, two or more transactions conflict if they access the same t-variable in a conflicting way, i.e., if at least one of those accesses is a write operation. (It is worth noting that the notion of a conflict can be easily generalized to t-variables with arbitrary operations, and to arbitrary mappings between t-variables and locks that may allow *false* conflicts. We discuss this in Section 7.)

Strong progressiveness is not the strongest possible progress property. The strongest one, which requires that no transaction is ever forcefully aborted, cannot be implemented without throttling significantly the parallelism between transactions, and is thus impractical in multiprocessor systems.

Strong progressiveness, however, still gives a programmer the following important advantages. First, it guarantees that if two independent subsystems of an application do not share any memory locations (or t-variables), then their transactions are completely isolated from each other (i.e., a transaction executed by a subsystem A does not cause a transaction in a subsystem B to be forcefully aborted). Second, it avoids "spurious" aborts: the cases when a transaction can abort are strictly defined. Third, it ensures global progress for single-operation transactions, which is important when non-transactional accesses to t-variables are encapsulated into transactions in order to ensure strong atomicity [3]. Finally, it ensures that processes are able to eventually communicate via transactions (albeit in a simplified manner—through a single t-variable at a time). Nevertheless, one can imagine many other reasonable progress properties, for which strong progressiveness can be a good reference point.

More precisely, let H be any history of a TM and T_k be any transaction in H. We say that T_k is *forcefully aborted* in H, if T_k is aborted in H and there is no invocation of operation $tryA_k$ in H. We denote by $WSet_H(T_k)$ and $RSet_H(T_k)$ the sets of t-variables on which T_k executed, respectively, a *write* or a *read* operation in H. We denote by $RWSet_H(T_k)$ the union of sets $RSet_H(T_k)$ and $WSet_H(T_k)$, i.e., the set of t-variables accessed (read or written) by T_k in history H. We say that two transactions T_i and T_k in H *conflict on a t-variable* x, if (1) T_i and T_k are concurrent in H, and (2) either x is in $WSet_H(T_k)$ and in $RWSet_H(T_i)$, or x is in $WSet_H(T_i)$ and in $RWSet_H(T_k)$. We say that T_k *conflicts with a transaction* T_i in H if T_i and T_k conflict in H on some t-variable.

Let H be any history, and T_i be any transaction in H. We denote by $CVar_H(T_i)$ the set of t-variables on which T_i conflicts with any other transaction in history H. That is, a t-variable x is in $CVar_H(T_i)$ if there exists a transaction $T_k \in H$, $k \neq i$, such that T_i conflicts with T_k on t-variable x.

Let Q be any subset of the set of transactions in a history H. We denote by $CVar_H(Q)$ the union of sets $CVar_H(T_i)$ for all $T_i \in Q$.

Let $CTrans(H)$ be the set of subsets of transactions in a history H, such that a set Q is in $CTrans(H)$ if no transaction in Q conflicts with a transaction *not* in Q. In particular, if T_i is a transaction in a history H and T_i does not conflict with any other transaction in H, then $\{T_i\} \in CTrans(H)$.

Definition 3 *A TM implementation M is* strongly progressive, *if in every history H of M the following property is satisfied: for every set $Q \in CTrans(H)$, if $|CVar_H(Q)| \leq 1$, then some transaction in Q is not forcefully aborted in H.*

4 Verifying Strong Progressiveness

Verifying that a given TM implementation M ensures a given property P might often be difficult as one has to reason about a large number of histories involving an arbitrary number of transactions accessing an arbitrary number of t-variables. This complexity is greatly reduced if one can reduce the verification task to some small subset of histories of M, e.g., involving a limited number of t-variables or transactions. This approach has been used, e.g., in [8] to automatically check opacity, obstruction-freedom, and lock-freedom of TMs that feature certain symmetry properties.

In this section, we show how to reduce the problem of proving strong progressiveness of histories with arbitrary numbers of transactions and t-variables to proving a simple property of each individual (logical) try-lock used in those histories. Basically, we show that if a TM implementation M uses try-locks, or if one can assign "logical" try-locks to some parts of the algorithm of M, and if each of those try-locks is strong, then M ensures strong progressiveness. Unlike in [8], we do not assume any symmetry properties of a TM. Our result is thus complementary to that of [8], not only because it concerns a different property, but also because it uses a different approach.

Our reduction theorem is general as it encompasses lock-based TMs that use invisible reads, i.e., in which readers of a t-variable are not visible to other transactions, as well as those that use visible ones. We show also how the theory presented here can be used to prove strong progressiveness of TL2, TinySTM, RSTM, and McRT-STM. Finally, we point out one of the ambiguities of ensuring strong progressiveness with visible reads.

4.1 Reduction Theorem

Let M be any TM implementation, and E be any implementation history of M. Let E' be any implementation history that is obtained from E by inserting into E any number of invocations and responses of operations of a try-lock L_x for every t-variable x. We say that E' is a *strong try-lock extension* of E, if the following conditions are satisfied in E':

STLE1. For every t-variable x, $E'|L_x$ is a valid history of a strong try-lock object;

STLE2. For every process p_i and every t-variable x, if, at some time t, p_i invokes *trylock* on L_x or p_i holds L_x, then p_i executes at t in E' a transaction T_k such that $x \in WSet_{E'}(T_k)$;

STLE3. For every process p_i and every transaction $T_k \in E'|p_i$, if T_k is forcefully aborted in E', then either (1) p_i while executing T_k is returned *false* from every operation *trylock* on some try-lock L_x, or (2) there is a t-variable $x \in RSet_{E'}(T_k)$, such that some process other than p_i holds L_x at some point while p_i executes T_k but before T_k acquires L_x (if at all).

Theorem 4 *For any TM implementation M, if there exists a strong try-lock extension of every implementation history of M, then M is strongly progressive.*

Proof. Assume, by contradiction, that there exists a TM implementation M, such that some implementation history E of M has a strong try-lock extension E', but E violates strong progressiveness. This means that there is a set Q in $CTrans(E)$, such that $|CVar_E(Q)| \leq 1$ and every transaction in Q is forcefully aborted in E. Recall that Q is a subset of transactions, such that no transaction in Q has a conflict with a transaction outside of Q.

Assume first that $CVar_E(Q) = \emptyset$. But then no transaction in set Q has a conflict, and so, by STLE1–2, no transaction in Q can fail to acquire a try-lock, or read a t-variable x such that try-lock L_x is held by a concurrent transaction. Hence, by STLE3, no transaction in Q can be forcefully aborted—a contradiction.

Let x be the t-variable that is the only element of set $CVar_E(Q)$. Note first that if a transaction T_k in Q invokes operation *trylock* on some try-lock L_y (where y is a different t-variable than x) then, by STLE2, no other transaction concurrent to T_k invokes *trylock* on L_y or reads t-variable y. This is because no transaction in Q conflicts on a t-variable different than x.

Assume first that no transaction in set Q acquires try-lock L_x. But then, by STLE1–3, no transaction in Q can be forcefully aborted—a contradiction.

Let T_k be the first transaction from set Q to acquire try-lock L_x. By STLE3, and because T_k is forcefully aborted, there is a transaction T_i that holds L_x after T_k starts and before T_k acquires L_x. Clearly, by STLE2, x must be in $WSet_E(T_i)$, and so T_i must be in set Q. But then T_i acquires L_x before T_k—a contradiction with the assumption that T_k is the first transaction from set Q to acquire L_x. □

4.2 Examples

We show here how our reduction theorems can be used to prove the strong progressiveness of TL2, TinySTM, RSTM (one of its variants), and McRT-STM. None of those TM implementations explicitly use try-locks, and so we need to show which parts of their algorithms correspond to operations on "logical" try-locks for respective t-variables. We assume the use of a simple contention manager that makes each transaction that encounters a conflict abort itself. Such a contention manager (possibly with a back-off protocol) is usually the default one in word-based TMs. We also assume that the mapping between t-variables and locks is a one-to-one function (which is the default in RSTM). This assumption is revisited in Section 7.

TL2. This TM uses commit-time locking and deferred updates. That is, locking and updating t-variables is delayed until the commit time of transactions. The TL2 algorithm is roughly the following (for a process p_i executing a transaction T_k):

1. When T_k starts, p_i reads the *read timestamp* of T_k from a global counter C.

2. If T_k reads a t-variable x, p_i checks whether x is not locked and whether the version number of x is lower or equal to the read timestamp of T_k. If any of those conditions is violated then T_k is aborted.

3. Once T_k invokes $tryC_k$, p_i first tries to lock all t-variables that were written to by T_k. Locking of a t-variable x is done by executing a compare-and-swap (CAS) operation on a memory word $w(x)$ that contains, among other information, a *locked* flag. If p_i successfully changes the *locked* flag from *false* to *true*, then p_i becomes the exclusive owner of x and can update x. If CAS fails, however, T_k is aborted.

4. Once all t-variables written to by T_k are locked, p_i atomically increments and reads the value of the global counter C. The read value is the *write timestamp* of T_k.

5. Next, p_i validates transaction T_k by checking, for every t-variable x read by T_k, whether x is not locked by a transaction other than T_k and whether the version number of x is lower or equal to the read timestamp of T_k. Again, if any of those conditions is violated then transaction T_k is aborted (and its locks released).

6. Then, p_i updates all the states of the locked t-variables with the values written by T_k and the write timestamp of T_k.

7. Finally, T_k releases all the locked t-variables.

It is easy to assign logical try-locks to the above algorithm of TL2, i.e., to build a try-lock extension of every implementation history E of TL2. Basically, we put an invocation and response of operation *trylock* on a try-lock L_x around any CAS operation that operates on the *locked* flag of any t-variable x. The response is *true* if CAS succeeds, and *false* otherwise. We also put an invocation and response of operation *unlock* on L_x around the write operation that sets the *locked* flag of x to *false*. It is straightforward to see that this way we indeed obtain a valid try-lock extension of any implementation history E of TL2:

1. Property STLE1 is ensured because a CAS on a word $w(x)$ can fail only when some other CAS on $w(x)$ already succeeded, and once a CAS on $w(x)$ succeeds, no other CAS on $w(x)$ can succeed until the *locked* flag is reset. Hence, the single CAS operation indeed implements a strong try-lock.

2. Property STLE2 is ensured because a transaction T_i invokes CAS on a word $w(x)$ only when (1) T_i wrote to t-variable x, and (2) T_i is in its commit phase.

3. To prove that TL2 ensures property STLE3, consider any forcefully aborted transaction T_k executed by some process p_i (in some implementation history E of TL2). Assume first that a CAS operation executed by T_k (i.e., by p_i while executing T_k) on some word $w(x)$ fails. But then (1) T_k could not have locked try-lock L_x before, and (2) T_k is immediately aborted afterwards. Hence, property STLE3 is trivially ensured. This means that T_k reads some t-variable x and either (1) $w(x)$ has the *locked* flag set to *true* when T_k reads x (and $w(x)$ is not locked by T_k), or (2) the version number of x is larger than the read timestamp of T_k. In case (1) property STLE3 is trivially ensured. Assume then case (2). This means that some transaction T_m that has a write timestamp greater than the read timestamp of T_k wrote to x either (a) before T_k read x, or (b) after T_k read x and before T_k locked $w(x)$. But then T_m must have acquired its write timestamp, while holding try-lock L_x, after T_k acquired its read timestamp and before T_k locked L_x (if at all). Hence, STLE3 is ensured.

We thus obtain the following theorem:

Theorem 5 *TL2 (with a one-to-one t-variable to try-lock mapping) is strongly progressive.*

TinySTM. There are two major differences with TL2. First, TinySTM locks a t-variable x already inside any *write* operation on x, i.e., locking is not delayed until the commit time of transactions. Second, if a transaction T_k reads a t-variable x that has a version number higher than the read timestamp of T_k, then T_k tries to validate itself to avoid being aborted, instead of aborting itself immediately. TinySTM uses CAS for locking, in the same way as TL2. Hence, we can insert the invocations and responses of operations on logical try-locks into any implementation history of TinySTM in the same way as for TL2.

It is worth noting, however, that the overflow handling mechanism, which can be turned on at compile time, breaks strong progressiveness in very long histories. As we discuss in Section 6.3, this mechanism is necessary to overcome the complexity lower bound and still guarantee correctness. However, strong progressiveness is still ensured in histories with the number of transactions lower than the maximum value of the t-variable version number, or between version number overflows.

Theorem 6 *TinySTM (with the overflow handling mechanism turned off, and with a one-to-one t-variable to try-lock mapping) is strongly progressive.*

RSTM. This TM is highly configurable: currently there are four TM backends to choose from, and each has a number of configuration options. The two backends that are of interest here are *LLT* and *RedoLock*. LLT is virtually identical to TL2. RedoLock has object-level lock granularity. That is, transactions conflict if they access (in a conflicting way) the same object, not necessarily the same memory location (i.e., t-variables in RSTM are objects, not single memory words as in TL2 and TinySTM). However, the algorithm of RedoLock is, depending on the configuration option, similar to either TL2 or TinySTM. The main difference is in the validation heuristic that decides when a transaction needs to validate its read set, but this does not impact strong progressiveness (the heuristic does not by itself abort any transaction—it just determines when to validate the read set of a transaction). Like in TL2 and TinySTM, RedoLock uses CAS for locking, and so the same technique as for TL2 and TinySTM can be used to prove that RSTM with RedoLock backend is strongly progressive.

Theorem 7 *RSTM with the RedoLock backend is strongly progressive.*

McRT-STM. The algorithm of McRT-STM (as described in [2]) is essentially the same as the one of TinySTM, except that McRT-STM does not validate reads until the commit time of a transaction (and so the timestamp-based read validation technique is not necessary). McRT-STM also does not handle timestamp overflows. Hence, as McRT-STM uses CAS for locking, it is immediate that McRT-STM is strongly progressive.

Theorem 8 *McRT-STM is strongly progressive.*

Visible reads. It may seem that the simplest way of implementing a strongly progressive TM that uses visible reads is to use read-write try-locks. Then, if a transaction T_i wants to read a t-variable x, T_i must first acquire a shared (read) try-lock on x, and if T_i wants to write to x, T_i must acquire an exclusive (write) try-lock on x. However, this simple algorithm does not ensure strong progressiveness, even if the read-write try-locks are (in some sense) strong. Consider transactions T_i and T_k that read a t-variable x. Clearly, both transaction acquire a shared lock on x. But then, if both T_i and T_k want to write to x, it may happen that both get aborted. This is because a transaction T_k cannot acquire an exclusive try-lock on x if any other transaction holds a shared try-lock on x.

A simple way to implement a strongly progressive TM with invisible reads is to use (standard) try-locks. Then, only the writing to a t-variable x requires acquiring a try-lock on x. A transaction that wants to reads x simply adds itself to the list of readers of x (if the try-lock of x is unlocked). This list, however, is not used to implement a read-write try-lock semantics, but to allow a transaction that writes to x to invalidate and abort all the current readers of x. Such a TM can be verified to be strongly progressive using our reduction theorem. A separate reduction theorem, based on read-write try-locks, is thus not necessary, and would probably be incorrect (trying to provide such a theorem allowed us to discover this ambiguity).

5 The Power of a Lock-Based TM

In this section, we use our semantics to determine the computational power of lock-based TMs. We use the notion of *consensus number* [13] as the metric of power of an object. The consensus number of an object x is defined as the maximum number of processes for which one can implement a wait-free *consensus* object using any number of instances of x (i.e., objects of the same type as x) and registers. A consensus object, intuitively, allows processes to agree on a single value chosen from the values those processes have proposed. More formally, a consensus object implements a single operation: *propose*(v). When a process p_i invokes *propose*(v), we say that p_i *proposes* value v. When p_i is returned value v' from *propose*(v), we say that p_i *decides* value v'. Every consensus object ensures the following properties in every execution: (1) no two processes decide different values (agreement), and (2) every value decided is a value proposed by some process (validity).

According to [13], if an object x has consensus number k, then one cannot implement x using objects with consensus number lower than k. For example, a queue and test-and-set have consensus number 2, and so they cannot be implemented from only registers (which have consensus number 1).

We prove here that the consensus number of a strongly progressive TM is 2. We do so in the following way. First, we prove that a strongly progressive TM is computationally equivalent to a strong try-lock. That is, one can implement a strongly progressive TM from (a number of) strong try-locks and registers, and vice versa. Second, we determine that the consensus number of a strong try-lock is 2.

The equivalence to a strong try-lock is interesting in its own right. It might also help proving further impossibility results as a strong try-lock is a much simpler object to reason about than a lock-based TM.

5.1 Equivalence between Lock-Based TMs and Try-Locks

To prove that a strongly progressive TM is (computationally) equivalent to a strong try-lock, we exhibit two algorithms: Algorithm 1 that implements a strong try-lock from a strongly progressive TM object and a shared memory register, and Algorithm 2 that implements a strongly progressive TM from a number of strong try-locks and registers. Both algorithms are not meant to be efficient or practical: their sole purpose is to prove the equivalence result.

The intuition behind Algorithm 1 is the following. We use an unbounded number of binary t-variables $x_1, x_2, \ldots$ (each initialized to *false*) and a single register V holding an integer (initialized to 1). If the value of V is v, then the next operation (*trylock* or *unlock*) will use t-variable x_v. If x_v equals *true*, then the lock is locked. A process p_i acquires the lock when p_i manages to execute a transaction T_k that changes the value of x_v from *false* to *true*. Then, p_i releases the lock by incrementing the value of register V, so that $x_{v'} = \mathit{false}$ where v' is the new value of V. (Note that incrementing V in two steps is safe here, as only one process—the one that holds the lock—may execute lines 2–12 at a time.) The implemented try-lock is strong because whenever several processes invoke *trylock*, at least one of those processes will commit its transaction (as the TM is strongly progressive) and acquire the try-lock.

Lemma 9 *Algorithm 1 implements a strong try-lock.*

Proof. We need to show that Algorithm 1, which we denote by A, is wait-free, ensures mutual exclusion, and implements a try-lock that is strong.

First, because there is no loop in A, and because both the TM object M and the register V are wait-free, algorithm A implements a wait-free object.

To prove mutual exclusion, observe that if several processes invoke operation *trylock* implemented by A and read the same value v in line 2, then, because TM object M ensures opacity, only one of them can commit a transaction that changes the value of t-variable x_v from *false* to *true*. Hence, only one of those processes can return *true* from the operation, i.e., acquire the try-lock. Observe also that only a process that holds the try-lock and then invokes operation *unlock* can change the value of register V. Hence, mutual exclusion is ensured.

Algorithm 1: An implementation of a strong try-lock from a strongly progressive TM object (k is a unique transaction identifier generated for every operation call)

```
uses: M—TM object, x1, x2, ...—binary t-variables,
      V—register
initially: x1, x2, ... = false, V = 1
1  operation trylock
2      v ← V.read;
3      locked ← M.tread_k(x_v);
4      if locked = A_k or locked = true then return false;
5      s ← M.twrite_k(x_v, true);
6      if s = A_k then return false;
7      s ← M.tryC_k;
8      if s = A_k then return false;
9      else return true;

10 operation unlock
11     v ← V.read;
12     V.write(v + 1);
13     return ok;
```

If a process p_i acquires a try-lock L implemented by A, then, by mutual exclusion, no process can acquire L and, a fortiori, invoke *unlock* on L until p_i invokes *unlock* on L. However, the operation *unlock* of p_i is not visible to other processes until p_i changes the value of V in line 12. Hence, only one process can execute lines 11–12 at any time, and so incrementing V in those lines is atomic.

This means that if L is unlocked and the value of V is v then $x_v = \mathit{false}$. Thus, if L is unlocked and several processes invoke *trylock* on L, then, by strong progressiveness of M, one of them, say p_i, observes in a transaction T_k that $x_v = \mathit{false}$, sets x_v to *true*, and commits T_k. Hence, p_i acquires L, and so L is a strong try-lock. □

The intuition behind Algorithm 2 is the following. We use a typical two-phase locking scheme with eager updates, optimistic (invisible) reads, and incremental validation (this can be viewed as a simplified version of TinySTM that explicitly uses strong try-locks). Basically, whenever a transaction T_i invokes operation *write* on a t-variable x for the first time, T_i acquires the corresponding try-lock L_x (line 13) and marks x as locked (line 21). Then, T_i may update the state of x in $TVar[x]$ any number of times. The original state of x is saved by T_i in $oldval[x]$, so that if T_i aborts then all the updates of t-variables done by T_i can be rolled back (line 39). If, at any time, T_i fails to acquire a try-lock, T_i aborts. This ensures freedom from deadlocks.

If T_i invokes operation *read* on a t-variable y that T_i has not written to before, T_i reads the current value of y (line 2) and *validates* itself (function *validate*). Validation ensures that none of the t-variables that T_i read so far has changed or has been locked, thus preventing T_i from having an inconsistent view of the system. If validation fails, T_i is aborted. Because values written to any

Algorithm 2: An implementation of a strongly progressive TM from strong try-locks and registers

```
   uses: L_x—strong try-lock (for each t-variable x),
         TVar—array of registers (other variables are
         local)
   initially: TVar[x] = (0, 0, false) for each t-variable x,
              rset = wset = ∅
1  operation tread_k(x)
2      (v, ts, locked) ← TVar[x].read;
3      if x ∈ wset then return v;
4      if x ∉ rset then
5          readts[x] ← ts;
6          rset ← rset ∪ {x};
7      if (not validate) or locked then
8          abort;
9          return A_k;
10     return v;
11 operation twrite_k(x, v)
12     if x ∉ wset then
13         locked ← L_x.trylock;
14         if not locked then
15             abort;
16             return A_k;
17     (v', ts, locked) ← TVar[x].read;
18     if x ∉ wset then
19         oldval[x] ← v';
20         wset ← wset ∪ {x};
21     TVar[x].write(v, ts, true);
22     return ok;
23 operation tryC_k
24     if not validate then
25         abort;
26         return A_k;
27     for x ∈ wset do
28         (v, ts, locked) ← TVar[x].read;
29         TVar[x].write(v, ts + 1, false);
30         L_x.unlock;
31     wset ← rset ← ∅;
32     return C_k;
33 operation tryA_k
34     abort;
35     return A_k;
36 function abort
37     for x ∈ wset do
38         (v, ts, locked) ← TVar[x].read;
39         TVar[x].write(oldval[x], ts, false);
40         L_x.unlock;
41     wset ← rset ← ∅;
42 function validate
43     for x ∈ rset do
44         (v, ts, locked) ← TVar[x];
45         if (locked and x ∉ wset) or ts ≠ readts[x] then
46             return false;
47     return true;
```

t-variable are not guaranteed to be unique, and because, in our simplified model, a try-lock does not have an operation that would read its state, we store with the state of each t-variable x a (unique) timestamp (version number) of x and a *locked* flag that is set to *true* if x is being written to by some transaction. The timestamps and *locked* flags are used for validation.

To commit a transaction T_i, the algorithm first validates T_i (line 24). Then, for each t-variable x written to by T_i, the timestamp of x is incremented by 1, the *locked* flag of x is set to *false* (line 29), and finally the try-lock L_x of x is unlocked (line 30). Aborting T_i requires rolling back all the updates done by T_i (line 39) and unlocking all the try-locks acquired by T_i (line 40).

Lemma 10 *Algorithm 2 implements a strongly progressive TM.*

Proof. Denote Algorithm 2 by A, and by M—an object implemented by A. Observe first that A is wait-free, because it contains no unbounded loops or waiting statements and because all the objects (try-locks and registers) used by A are wait-free. It is also straightforward to see that every implementation history E of A is its own strong try-lock extension, i.e., E ensures properties STLE1–4. Hence, M is strongly progressive.

Let us prove that A ensures opacity. Let T_i be any transaction, and x—any t-variable. We say that T_i: (1) *reads* x if T_i executes line 2 for x and does not abort after the subsequent validation in line 7, (2) *locks* x if T_i executes line 21 for x, (3) *commits* x if T_i executes line 29 for x, and (4) *aborts* x if T_i executes line 39 for x.

Observe first that if T_i writes value v to x, then the subsequent *read* operations of T_i on x will return v. Also, if T_i locks x, then no other transaction can read any value from x until T_i commits or aborts x, and so only the last value written to x by T_i may be read by other transactions. Hence, we can consider only those histories of A in which a transaction T_i that writes to a t-variable x does not invoke any further operations on x.

Let E be any such implementation history of A. Let T_i be any transaction in E. Whenever T_i reads a t-variable x, T_i re-reads (validates) all the t-variables that T_i read so far, including x. Hence, T_i always observes a consistent state of t-variables: if any validation fails, T_i is immediately aborted without being returned the inconsistent new value. This means that *read* operations of T_i are atomic: they appear as if they took place instantaneously at some time t in E. Moreover, this time t must be somewhere within the lifespan of T_i, because T_i observes updates of transactions that committed before T_i started.

If T_i is a transaction in E that has not committed any t-variable, then A ensures that no value written to any t-variable by T_i is visible to other transactions. That is because of the following. First, a transaction T_k may read a t-variable x only if T_k reads in line 2 a value with *locked* field set to *false*. Second, whenever T_i writes to a t-variable x, T_i always writes to $TVar[x]$ a value with *locked* field set to *true* (line 21), and then, inside function *abort*, T_i restores the value of $TVar[x]$ to the original one, with *locked* field set to *false*(line 39).

As the reads of every transaction are atomic, and the writes of every transaction that has not committed any t-variable are not visible to other transactions, we can focus only on those transactions in E that have committed at least one t-variable.

Let T_i be any transaction in E that has committed at least one t-variable. We denote by $t(T_i)$ the longest period (t_1, t_2), such that T_i does not read or lock any t-variable after t_1, and T_i has not invoked function *validate* in line 24 by t_2. If T_i reads x, then no transaction can commit x in $t(T_i)$; otherwise, the validation of T_i that follows $t(T_i)$ would fail and T_i would not commit any t-variable. This also means that no transaction T_k other than T_i that commits x in E can lock x during $t(T_i)$, and if T_k locks x before $t(T_i)$, then T_k must also commit x before $t(T_i)$; otherwise, T_i would observe in its validation phase after $t(T_i)$ that either x is locked or x has been committed, and so T_i would abort. If T_i locks x, then no transaction can lock or commit x in $t(T_i)$, because try-lock $L(x)$ is held by T_i during $t(T_i)$.

Therefore, if a transaction T_i (that commits some t-variable) reads or commits a t-variable x, and a transaction T_k commits x, then $t(T_i)$ and $t(T_k)$ do not overlap. Hence, T_i appears to execute atomically either before T_k or after T_k. Thus, transactions that commit at least one t-variable are also atomic.

□

From Lemma 9 and Lemma 10, we immediately obtain the following result (recall that an object x is (computationally) equivalent to an object y, if y can be implemented from any number of instances of x and registers, and x can be implemented from any number of instances of y and registers):

Theorem 11 *Every strongly progressive TM is equivalent to a strong try-lock.*

5.2 Consensus Number of Strong Try-Locks

To prove that the consensus number of a strong try-lock is 2, we show that (1) a strong try-lock can implement consensus in a system of 2 processes, and (2) there is no algorithm that implements consensus using (any number of) strong try-locks and registers in a system of 3 (or more) processes.

Algorithm 3 shows an implementation of consensus for two processes (p_1 and p_2) using a single strong try-lock (L) and two registers (V_1 and V_2). The process p_i that acquires L is the winner: the value proposed by p_i, and written by p_i to register V_i, is decided by both p_1 and p_2. Because L is a strong try-lock, if both processes concurrently execute operation *propose*, at least one of them acquires L. Because no process ever unlocks L, at most one process acquires L. Hence, exactly one process is the winner.

Algorithm 3: An implementation of wait-free consensus from a strong try-lock in a system of 2 processes (code for process p_i, $i = 1, 2$)

```
uses: L—strong try-lock, V1, V2—registers

operation propose(v)
    Vi.write(v);
    locked ← L.trylock;
    if locked then return v;
    else return V(3−i).read;
```

Lemma 12 *Algorithm 3 implements wait-free consensus in a system of 2 processes.*

Proof. Denote Algorithm 3 by A. First, A is a wait-free implementation, because it does not contain any loop or waiting statement, and the base objects used by A (L, V_1, and V_2) are wait-free.

Second, a value returned by operation *propose* executed by a process p_i may be either the value proposed by p_i (in which case validity is trivially ensured) or the value of register $V_{(3-i)}$. The latter case is possible only if p_i is returned *false* from operation *trylock* on L, and this, in turn, is only possible if process $p_{(3-i)}$ is concurrently executing *trylock* on L. Then, however, $p_{(3-i)}$ must have already written its proposed value to $V_{(3-i)}$, and so also in this case validity is ensured at p_i.

Finally, assume, by contradiction, that there is some implementation history E of A in which agreement is violated. That is, process p_1 decides value v_1 and p_2 decides value $v_2 \neq v_1$. But then both processes must have either returned *true* or *false* from operation *trylock* on L. If p_1 and p_2 both return *true* from *trylock*, then both processes hold L, which violates mutual exclusion. If p_1 and p_2 both return *false* from *trylock*, then, as there is no other invocation of *trylock* on L, this means that try-lock L is not strong. Hence, agreement must be ensured. □

To prove that there is no algorithm that implements consensus using strong try-locks and registers in a system of 3 (or more) processes, we show in Algorithm 4 that a strong try-lock can be implemented from a single test-and-set object.[6] Because a test-and-set object has consensus number 2, the algorithm proves that a strong try-lock cannot have consensus number higher than 2. Note that the presented algorithm is a non-blocking version of a simple and well-known TAS lock [24]. The following lemma is thus trivial to verify:

Lemma 13 *Algorithm 4 implements a strong try-lock.*

From Lemma 12 and Lemma 13, we immediately obtain the following result:

Theorem 14 *A strong try-lock has consensus number 2.*

[6] A test-and-set object has two operations: *test-and-set*, which atomically reads the state of the object, sets the state to *true*, and returns the state read, and *reset*, which sets the state to *false*.

Algorithm 4: An implementation of a strong try-lock from a test-and-set object

```
uses: S—test-and-set object
initially: S = false

operation trylock
    locked ← S.test-and-set;
    return ¬ locked;

operation unlock
    S.reset;
```

Hence, by Theorem 11 and Theorem 14, the following theorem holds:

Theorem 15 *Every strongly progressive TM has consensus number 2.*

Corollary 16 *There is no algorithm that implements a strongly progressive TM using only registers.*

5.3 Weakening Strong Progressiveness

Interestingly, nailing down precisely the progress property of a lock-based TM also helps consider alternative semantics and their impacts. We discuss here how one has to weaken the progress semantics of a lock-based TM so that it could be implemented with registers only. We define a property called *weak progressiveness* that enables (lightweight) TM implementations with consensus number 1.

Intuitively, a TM is weakly progressive if it can forcefully abort a transaction T_i only if T_i has a conflict with another transaction. More precisely:

Definition 17 *A TM implementation M is* weakly progressive, *if in every history H of M the following property is satisfied: if a transaction $T_i \in H$ is forcefully aborted, then T_i conflicts with some transaction in H.*

We correlate this notion with the concept of a *weak try-lock*: a try-lock which operation *trylock* executed by a process p_i may always return *false* if another process is concurrently executing *trylock* on the same try-lock object. That is, p_i is guaranteed to acquire a weak try-lock L only if L is not locked and no other process tries to acquire L at the same time. More precisely:

Definition 18 *We say that a try-lock L is* weak *if L has the following property: if a process p_i invokes trylock on L at some time t, L is not locked at t, and no process other than p_i executes operation trylock on L at time t or later, then p_i is returned true.*

While we do not know of any existing implementation of a weak try-lock, such an implementation can be easily obtained from several well-known (blocking) mutual exclusion algorithms, e.g., those proposed in [18] that ensure at least the *shutdown safety* property introduced in the same paper.

An example implementation of a weak try-lock using only registers, similar in concept to some of the lock implementations in [18], is given in Algorithm 5. The intuition behind the algorithm is the following. If a process p_i invokes operation *trylock* on a try-lock L implemented by the algorithm, p_i first checks whether any other process holds L (lines 2–3). If not, p_i announces that it wants to acquire L by setting register $R[i]$ to 1 (line 4). Then, p_i checks whether it is the only process that wants to acquire L (lines 5–6). If yes, then p_i acquires L (returns *true*). Otherwise, p_i resets $R[i]$ back to 0 (so that future invocations of *trylock* may succeed) and returns *false*. Clearly, if two processes execute *trylock* in parallel, then both can reach line 6. However, then at least one of them must observe that more than one register in array L is set to 1, and return *false*.

Lemma 19 *Algorithm 5 implements a weak try-lock.*

Proof. Denote Algorithm 5 by A, and by L—a try-lock object implemented by A. First, it is straightforward to see that A is wait-free: it does not have any loops or waiting statements and all base objects used by A are wait-free.

Assume, by contradiction, that A does not ensure mutual exclusion. Hence, there is an implementation history E of A in which some two processes, say p_i and p_k, hold L at some time t. Consider only the latest *trylock* operations of p_i and p_k on L before t. Both of those operations must have returned *true*. Process p_i observes that $R[k] = 0$ in line 5, and so p_i reads $R[k]$ before p_k writes 1 to $R[k]$ in line 4. Hence, p_k reads $R[i]$ (in line 5) after p_i writes 1 to $R[i]$. Thus, p_k reads that $R[i]$ and $R[k]$ equal 1 and returns *false* in line 6—a contradiction.

It is easy to see that, for any process p_i, if $R[i] = 1$ then either p_i holds L or p_i is executing operation *trylock* on L. Hence, if a process p_i returns *false* from *trylock*, then either L is held by another process or another process is executing *trylock* concurrently to p_i. This means that L is a weak try-lock. □

From Lemma 19, we obtain the following result:

Theorem 20 *A weak try-lock has consensus number 1.*

It is straightforward to see that using weak try-locks instead of strong ones in the TM implementation shown in Algorithm 2 gives a TM that ensures weak progressiveness. Hence, by Theorem 20, we immediately prove the following result:

Theorem 21 *Every weakly progressive TM has consensus number 1.*

6 Performance Trade-Off

We prove that the space complexity of every weakly (and, a fortiori, strongly) progressive TM that uses in-

Algorithm 5: An implementation of a weak try-lock using registers (code for process p_i)

```
uses: R[1,...,n]—array of registers
initially: R[k] = 0 for k = 1,...,n

operation trylock
    s ← getsum;
    if s > 0 then return false;
    R[i].write(1);
    s ← getsum;
    if s = 1 then return true;
    R[i].write(0);
    return false;

operation unlock
    R[i].write(0);
    return ok;

function getsum
    s ← 0;
    for k = 1 to n do s ← s + R[k].read;
    return s;
```

visible reads is at least exponential with the number of t-variables available to transactions. The invisible reads strategy is used by a majority of TM implementations [4, 20, 12, 2, 7] as it allows efficient optimistic reading of t-variables. Intuitively, if invisible reads are used, a transaction that reads a t-variable does not write any information to base objects. Hence, many processors can concurrently execute transactions that read the same t-variables, without invalidating each other's caches and causing high traffic on the inter-processor bus. However, transactions that update t-variables do not know whether there are any concurrent transactions that read those variables.

6.1 Semantics of Invisible Reads

We state our lower bound result assuming a simplified definition of the notion of invisible reads. This is sufficient for our lower bound proof, and is in agreement with what is ensured by various TM implementations [4, 20, 7]. Intuitively, we say that a TM implementation M uses invisible reads, if it does not modify the state of any base object when processing a *read* operation on any t-variable.

We capture this more precisely using the notion of a configuration. A *configuration* is the state of all base objects at a given point in time. Assuming that the initial state of base objects is fixed, and that base objects are deterministic, the configuration after any implementation history E can be precisely determined.

Let E be any implementation history of a TM. We define an *operation execution* of a process p_i in E to be any pair of (a) an invocation of operation *tread* or *twrite* and (b) the subsequent response of this operation in the sub-history $E|p_i$. If e is an operation execution of some process p_i in E, then every step in $E|p_i$ between the invocation and the response of e is said to be *corresponding to e*.

Definition 22 *A TM implementation M uses* invisible reads *if, for every implementation history E of M, no step corresponding to an execution of operation tread in E changes the configuration.*

6.2 The Lower Bound

The *size* of a t-variable or a base object x can be defined as the number of distinct, reachable states of x. In particular, if x is a t-variable or a register object, then the size of x is the number of values that can be written to x. For example, the size of a 32-bit register is 2^{32}.

Theorem 23 *Every weakly progressive TM implementation that uses invisible reads and provides to transactions N_s t-variables of size K_s uses $\Omega\left({K_s}^{N_s}/K_b\right)$ base objects of size K_b.*

Proof. Let M be any weakly progressive TM implementation that uses invisible reads and provides N_s t-variables of size K_s. Assume that M uses N_b base objects of size K_b. Clearly, if $K_b = \infty$ or $N_b = \infty$, the theorem trivially holds. Assume then, that K_b and N_b are finite numbers. Also, if $K_s = \infty$ or $N_s = \infty$, then one obviously needs either an infinite number of base objects, or a base object of infinite size to store the states of all the t-variables provided by I. In either case, the theorem trivially holds. Hence, assume that K_s and N_s are finite numbers.

Let $x_1, \ldots, x_{N_s}$ be the t-variables provided by M. For simplicity, but without loss of generality, assume that every t-variable x_k has the same domain of values D ($|D| = K_s$). Let $S = \{s_1, s_2, \ldots, s_L\}$ ($L = {K_s}^{N_s}$) be the set of all tuples $(v_1, \ldots, v_{N_s})$, where each value v_m, $m = 1, \ldots, N_s$, is in D. We say that a transaction T_k *writes tuple* $s \in S$, if T_k writes to every t-variable x_m, $m = 1, \ldots, N_s$, the m^{th} value from s.

Let U be the set of all implementation histories of M, in which process p_1 executes infinitely many transactions, each of which writes a tuple from set S and commits, and no other process takes steps. Note that no transaction can be forcefully aborted in any implementation history $E \in U$, because all transactions in E are executed by a single process, and so no two transactions in E are concurrent. Let Q be the set that contains a configuration after every complete prefix of every implementation history from set U. (A prefix E' of an implementation history E is complete if every transaction in E' is either committed or aborted.)

Consider a set of configurations $W \subseteq Q$, and two configurations c and c' in W. We write $c \rightarrow_W c'$, if there exists an implementation history $E \in U$, a complete prefix E_t of E, and a complete prefix E_f of E_t, such that the configuration after E_f is c, the configuration after E_t is c', and a configuration after every complete prefix of E_t that contains E_f is in set W.

Let s_i be a tuple in set S. We denote by $C^*(s_i)$ the set of configurations in Q that occur after any complete prefix E of any implementation history in U, such that the last transaction in E writes tuple s_i. Clearly, if E is a finite, complete implementation history of M the configuration after which is in $C^*(s_i)$, and E' is an extension of E, in which no transaction writes to any t-variable after E, then every read operation on a t-variable x_j invoked after E must return the j^{th} value from tuple s_i; otherwise the real-time ordering required by opacity would be violated.

Let $S_i = \{s_i, \ldots, s_L\} \subseteq S$, where $i = 1, \ldots, L$. We prove the following claim:

Claim 24 *There exist subsets Q_1, ..., Q_L of Q, such that $Q_L \subset \ldots \subset Q_1$, and the following conditions are satisfied by every set Q_k, $k = 1, \ldots, L$:*

1. *For every tuple $s \in S_k$, $C^*(s) \cap Q_k \neq \emptyset$,*
2. *For every tuple $s \in S - S_k$, $C^*(s) \cap Q_k = \emptyset$, and*
3. *For every pair of configurations $c, c' \in Q_k$, $c \rightarrow_{Q_k} c'$.*

Proof. Consider an implementation history $E \in U$ constructed in the following way (all transactions in E are executed by process p_1 and are committed; initially *round* $= 1$):

1. Transaction $T_{L,0}^{round}$ writes tuple s_L.
2. For *setnum* $\leftarrow L, \ldots, 1$ the following scenario is repeated:
   (a) Transaction $T_{setnum,1}^{round}$ writes tuple s_{setnum}. Denote by $c_{setnum,1}^{round}$ the configuration after $T_{setnum,1}^{round}$ is executed.
   (b) Let G be a finite sequence of tuples in set S_{setnum}, such that if a sequence of transactions is executed, each writing the subsequent tuple from G, then the resulting configuration is the same as $c_{setnum,1}^{round}$.
      - If no such sequence G exists, denote by f^{round} the current value of *setnum*, increase *round* by 1, and go back to step 1.
      - If such G exists, execute a sequence of transactions $T_{setnum,2}^{round}$, ..., $T_{setnum,m}^{round}$, $m = |G| + 1$, each writing the subsequent tuple from G. Clearly, the configuration after $T_{setnum,m}^{round}$ is executed is the same as $c_{setnum,1}^{round}$.

Note first that because process p_1 executes transactions in E alone, and M is weakly progressive, no transaction in E can be forcefully aborted. Thus, E indeed contains only committed transactions. Note also that in each round q (i.e., for each *round* $= q$) transactions write tuples only from set S_{f^q} (or S in the last round). That is because each set S_i is a superset of every set S_k, where $k = i+1, \ldots, L$, and $S_L = \{s_L\}$.

Let us show that E is finite, i.e., that the above algorithm always terminates. By contradiction, assume that E is infinite. Denote by $c_{L,k}^{round}$, $k = 0, 1, \ldots$, the configuration after transaction $T_{L,k}^{round}$ is executed in E. As the number of configurations is finite, there must be some two transactions $T_{L,0}^{q}$ and $T_{L,0}^{w}$ in E, $q < w$, such that the configurations $c_{L,0}^{q}$ and $c_{L,0}^{w}$ are the same. Let m, $q \leq m < w$, be a value for which f^m is minimal. Because f^m is a minimal value of $f^q, \ldots, f^{w-1}$, all transactions between $T_{L,0}^{q}$ and $T_{L,0}^{w}$ (including $T_{L,0}^{q}$ and $T_{L,0}^{w}$) write tuples from set S_{f^m}. Note also that, by the definition of value f^m, a sequence G of tuples could not be found in step 2b of the algorithm after transaction $T_{f^m,1}^{m}$ was executed.

Consider the sequence W of transactions executed after transaction $T_{f^m,1}^{m}$ and up to, and including, transaction $T_{L,0}^{w}$ in E. Sequence W, in which all transactions write tuples from set S_{f^m}, changes the configuration from $c_{f^m,1}^{m}$ to $c_{L,0}^{w} = c_{L,0}^{q}$. Hence, a sequence G of tuples that satisfies the condition in step 2b of the algorithm after transaction $T_{f^m,1}^{m}$ is executed exists, and we reach a contradiction.

Let t be the value of *round* at which the algorithm terminated. Let Q_k, $k = 1, \ldots, L$, be the set of configurations after every complete prefix of E that contains transaction $T_{L,0}^{t}$ and does not contain transaction $T_{k-1,1}^{t}$. It is straightforward to see that each set Q_k satisfies the conditions in the claim:

1. Every configuration $c_{m,1}^{t}$, $m = k, \ldots, L$, is in set Q_k and must be in set $C^*(s_m)$ because transaction $T_{m,1}^{t}$ writes tuple s_m.
2. No configuration in Q_k can be in any set $C^*(s_i)$, $i = 1, \ldots, k-1$, because no transaction after and including $T_{L,0}^{t}$ and before $T_{k-1,1}^{t}$ in E writes tuple s_i.
3. Let $s_{q_1}, \ldots, s_{q_m}$ be the sequence of respective tuples written by transactions executed after $T_{L,0}^{t}$ and before transaction $T_{k-1,1}^{t}$ in E. Let c be the configuration after a transaction T that writes tuple s_{q_k}, and c' be the configuration after a transaction T' that writes tuple s_{q_j} (we assume that both T and T' are between $T_{L,0}^{t}$ and $T_{k-1,1}^{t}$ in E). Clearly, both c and c' are in set Q_k. If T precedes T', then a sequence of transactions (executed by process p_1) that write, subsequently, tuples $s_{q_{j+1}}, \ldots, s_{q_k}$ changes the configuration from c to c'. If T' precedes T, then a sequence of transactions that write, subsequently, tuples $s_{q_{k+1}}, \ldots, s_m, s_{q_1}, \ldots, s_{q_j}$ changes the configuration from c to c'. In either case, we have that $c \rightarrow_{Q_k} c'$.

□

If c is a configuration and b is any base object, then we denote by $c(b)$ the state of b in configuration c. If c_1, ..., c_t are configurations, then $C(c_1, \ldots, c_t)$ denotes the set of configurations, such that $c \in C(c_1, \ldots, c_t)$ if, for every base object b, $c(b) \in \{c_1(b), \ldots, c_t(b)\}$. We prove the following claim:

Claim 25 *For every tuple $s_i \in S$, $i < L$, every $k \in \{i + 1, \ldots, L\}$, and every subset $\{c_1, \ldots, c_t\}$ of Q_k, $C(c_1, \ldots, c_t) \cap C^*(s_i) = \emptyset$.*

Proof. By contradiction, assume that there exists a tuple $s_i \in S$, a configuration $c \in C^*(s_i)$, a value $k \in \{i+1, \ldots, L\}$, and a subset $\{c_1, \ldots, c_t\}$ of Q_k, such that $c \in C(c_1, \ldots, c_t)$. Consider an infinite implementation history $E_k \in U$, in which process p_1 executes infinitely many transactions, such that (1) every configuration in Q_k occurs after infinitely many complete prefixes of E_k, and (2) there exists a finite prefix E_k^{p} of E_k such that the configuration after every prefix of E_k that contains E_k^{p} is in set Q_k. By Claim 24, such an implementation history E_k indeed exists. Let E'_k be an implementation history of M, such that $E'_k|p_1 = E_k$, in which process p_2 executes a single transaction T_0, started after E_k^{p}, that reads from all t-variables. We will lead to a contradiction by showing that there exists such an interleaving of steps of p_1 and p_2 in E'_k for which T_0 is not forcefully aborted before T_0 invokes $tryC_0$, and for which T_0 is returned values from tuple $s_i \notin S_k$. This means that opacity is violated in E'_k because s_i is not written by any transaction in E'_k after E_k^{p} (as $C^*(s_i) \cap Q_k = \emptyset$).

We construct E'_k in the following way. Before each step of p_2, we make p_1 execute and complete a number of transactions, such that p_2 always observes the states of base objects as in configuration c. More precisely, if p_2 is about to access a base object b in its next step, we make p_1 execute and complete transactions, until the system reaches a configuration c', such that $c'(b) = c(b)$. This is possible, because of the following:

1. By our assumption, for each base object b', there exists a configuration $c' \in \{c_1, \ldots, c_t\}$, such that $c'(b') = c(b')$.

2. The set $\{c_1, \ldots, c_t\}$ is a subset of Q_k, and every configuration in Q_k occurs infinitely many times in E_k.

3. Until p_2 changes the state of any base object, E'_k is indistinguishable for p_1 from E_k (i.e. ,p_1 executes the same steps and receives the same responses in E'_k as in E_k), and so p_1 changes the configuration in the same way in both E_k and E'_k.

4. E'_k is indistinguishable for p_2 from an implementation history in which p_2 executes steps alone, starting from configuration c (i.e., in which p_1 first executes a series of transactions, the last of which writes tuple s_i, leaving the system in configuration $c \in C^*(s_i)$, and then p_2 executes T_0 alone until $tryC_0$). Thus, as M is weakly progressive, T_0 cannot be forcefully aborted before $tryC_0$. Moreover, as M uses invisible reads, p_2 does not change the state of any base object until T_0 invokes $tryC_0$.

In E'_k, T_0 reads from all t-variables, and, as p_2 observes a configuration from set $C^*(s_i)$, T_0 is returned values from tuple s_i that is never written in E'_k after prefix E_k^{p}—a contradiction with opacity. □

If c is a configuration in $C^*(s_k)$, $k < L$, then we denote by $I_k(c)$ the set of pairs (b_l, v_q), where b_l is a base object and v_q is a state of b_l, such that (1) $c(b_l) = v_q$, and (2) $c'(b_l) \neq v_q$ for every $c' \in Q_{k+1}$.

Claim 26 *For every tuple s_k, $k < L$, and every configuration $c \in C^*(s_k)$, $I_k(c) \neq \emptyset$.*

Proof. The proof follows directly from Claim 25: if it was that $I_k(c) = \emptyset$ for some $k < L$ and $c \in C^*(s_k)$, it would mean that configuration c is also in set $C(c_1, \ldots, c_t)$, for some configurations $c_1, \ldots, c_t \in Q_{k+1}$. □

Claim 27 *For every $1 < k < L$, there is a configuration $c \in C^*(s_k)$ and a pair $(b_l, v_q) \in I_k(c)$, such that $(b_l, v_q) \notin I_m(c')$ for every $m < k$ and every $c' \in C^*(s_m)$.*

Proof. Consider any k, $1 < k < L$. By the properties of set Q_k, there exists a configuration $c \in C^*(s_k)$ that is also in Q_k. Furthermore, there is no $m < k$ for which $C^*(s_m) \cap Q_k \neq \emptyset$. Thus, $c \notin C^*(s_m)$ for every $m < k$. By Claim 26, $I_k(c) \neq \emptyset$. Let (b_l, v_q) be some element of $I_k(c)$. Clearly, $c(b_l) = v_q$. But for every $m < k$, every configuration $c' \in C^*(s_m)$ and every pair $(b'_l, v'_q) \in I_m(c')$, $c(b'_l) \neq v'_q$ because $c \in Q_k \subseteq Q_{m+1}$. Thus, $(b_l, v_q) \notin I_m(c')$. □

By Claim 26 and Claim 27, we have that there is at least as many unique pairs (b_l, v_q), where b_l is a base object and v_q is a state of b_l, as tuples in S. Thus, M needs at least $L/K_{\mathrm{b}} = {K_{\mathrm{s}}}^{N_{\mathrm{s}}}/K_{\mathrm{b}}$ base objects. □

6.3 Overcoming the Lower Bound

Our lower bound relies on three properties of a TM: weak progressiveness, operation wait-freedom, and invisible reads. It could seem that weakening (reasonably) any of those properties would allow overcoming the lower bound. We explain (informally) in the following paragraphs why this is not the case, and what has to be done in order for the lower bound not to hold.

Consider the following progress property, which is strictly weaker than weak progressiveness: if a transaction T_i is forcefully aborted, then there must be a transaction concurrent to T_i. We say that a TM that ensures this property is *non-trivial*—indeed, this seems like a basic requirement for a TM. However, non-trivial TMs do not overcome the complexity bound if they ensure operation wait-freedom and use invisible reads. Basically, in the proof of Theorem 23, transactions executed by processes p_1 and p_2 are not aware of any concurrent transactions, and so they will not be forcefully aborted in a non-trivial TM.

Consider the following liveness property, which we call *termination*: if a process p_i invokes an operation on a TM object, then p_i eventually returns from the operation. Clearly, termination is strictly weaker than wait-freedom. Consider a TM that ensures weak progressiveness and termination, and that uses invisible reads. Again, the complexity lower bound holds for such a TM: as in the proof of Theorem 23 process p_1 and p_2 is not aware of the operations executed by the other process, no process can block waiting for the other one to execute steps. Hence, in the particular execution used in the proof, termination would be sufficient.

Assume now that we allow a TM that uses invisible reads to update the state of a constant number of base objects in the first operation of every transaction, even if this operation is a *read*. We say then that such a TM uses *weak invisible reads*. Hence, each transaction is allowed to announce its start. This means that, in the proof of Theorem 23, process p_1 can be aware of transaction T_2 executed by process p_2. However, if the transactions executed by p_1 and p_2 access not all but almost all t-variables (all except for a constant number), then p_2 would not be allowed (in general) to forcefully abort its transactions, as there would be no guarantee that there is a conflict between those transactions and transaction T_2.

This means that to overcome the lower bound we need to weaken more than one property of the TM. For example, TinySTM can be compiled with an option to enable a mechanism that handles timestamp overflows. (Without such a mechanism TinySTM can violate opacity in very long executions, as can TL2 or the LLT backend of RSTM.) Then, TinySTM uses weak invisible reads and may periodically violate both strong progressiveness and operation wait-freedom. Roughly speaking, once a transaction overflows a version number of a t-variable x, all transactions that access x are aborted, and all transactions that start afterwards are blocked on a barrier. Once there is no running transaction, the version number of x can be reset and transactions can proceed. This means that TinySTM ensures strong progressiveness and operation wait-freedom between timestamp overflows, but when an overflow happens the TM becomes only non-trivial and its operation-level liveness is reduced to termination.

7 Concluding Remarks

The two major assumptions we made in this paper were that t-variables support only *read* and *write* operations, and that the mapping between t-variables and corresponding try-locks is a one-to-one relation. We discuss here how those assumptions can be relaxed (at the price of increasing the complexity of the model and definitions). We also discuss the problem of model checking TMs for strong progressiveness.

Arbitrary t-variables. Object-based TMs support t-variables of arbitrary type. However, most of them classify all the operations of t-variables as either read-only or update ones. In those cases, there is no need to extend our simplified model, because read-only operations are effectively *read*s, and update operations are effectively pairs of *read*s and *write*s.

We can, however, imagine a TM that exploits the commutativity relations between some operations of t-variables of any type. In this case, one can extend the model of a TM to allow for arbitrary operations on t-variables, and redefine the notion of a conflict. Indeed, operations that commute should not conflict. Consider for example a counter object and its operation *inc* that increments the counter and does not return any meaningful value. It is easy to see that there is no real conflict between transactions that concurrently invoke operation *inc* on the same counter: the order of those operations does not matter and is not known to transactions (it would be, however, if *inc* returned the current value of the counter).

Once the notion of a conflict is defined, our definitions of progress properties remain correct even for t-variables with arbitrary operations. If we assume that a TM must support t-variables with operations *read* and *write* (in addition to other t-variables), then also the consensus number and complexity lower bound results hold for those TMs. However, the question of how to verify strong progressiveness of TM implementations with arbitrary t-variables is an open problem.

Arbitrary t-variable to try-lock mappings. Many lock-based TMs employ a hash function to map a t-variable to the corresponding try-lock. It may thus happen that a false conflict occurs between transactions that access disjoint sets of t-variables, and so, a priori, strong progressiveness might be violated. However, it is easy to take the hash function h of a TM implementation M into account in the definition of strong progressiveness. Basically, when a transaction T_i reads or writes a t-variable x in a history H of M, we add to, respectively, the read set ($RSet_H(T_i)$) or the write set ($WSet_H(T_i)$) of T_i not only x, but also every t-variable y such that $h(x) = h(y)$. The definition of a conflict hence also takes into account false conflicts between transactions, and the strong progressiveness property can be ensured by M. (Such a property could be called *h-based strong progressiveness*.) It is important to note, however, that the hash function must be known to a user of a TM, or even provided by the user. Otherwise, strong progressiveness (and, for that matter, any other property that relies on the notion of a conflict) would no longer be visible, and very meaningful, to a user.

Model checking. While our reduction theorem simplifies proving strong progressiveness of a TM implementation, it might still be difficult to verify this property in an automatic manner. Indeed, even when verifying histories from the perspective of individual try-locks, we have to deal with an unbounded number of states. A solution would be to propose a reduction theorem along the lines of [8], assuming that a TM implementation has certain symmetry properties. Two problems arise then. First, one has to express those properties in the fine-grained model we use ([8] assumes operations like *validate* or *commit* to be atomic). Second, one has to prove that a given TM implementation ensures those properties, which is not always trivial (e.g., properties P6 and P7 in [8]). Both problems remain open.

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] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006.

[5] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[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)*, Feb 2008.

[8] R. Guerraoui, T. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008.

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

[10] 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.

[11] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, 1988.

[12] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, 2006.

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

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[15] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

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

[17] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

[18] L. Lamport. The mutual exclusion problem—part II: Statement and solutions. *Journal of the ACM*, 33(2), 1985.

[19] L. Lamport. On interprocess communication—part II: Algorithms. *Distributed Computing*, 1(2), 1986.

[20] 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)*, Jun 2006.

[21] 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.

[22] 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.

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

[24] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

[25] M. L. Scott. Sequential specification of transactional memory semantics. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.

[26] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2001.

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