Transactional Memory: Glimmer of a Theory (Invited Paper)

Rachid Guerraoui and Michał Kapałka

EPFL, Switzerland

Abstract. Transactional memory (TM) is a promising paradigm for concurrent programming. This paper is an overview of our recent theoretical work on defining a theory of TM. We first recall some TM correctness properties and then overview results on the inherent power and limitations of TMs.

1 Introduction

Multi-core processors are already common in servers, home computers, and laptops. 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) [1] is a promising technique to facilitate concurrent programming while delivering performance comparable to that of fine-grained locking implementations. In short, a TM allows concurrent threads of an application to communicate by executing lightweight, in-memory *transactions* [2]. 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.

The TM paradigm has raised a lot of hope for mastering the complexity of concurrent programming. The aim is to provide the programmer with an abstraction, i.e., the transaction, that makes concurrency as easy as with coarse-grained critical sections, while exploiting the underlying multi-core architectures as efficiently as hand-crafted fine-grained locking. It is thus not surprising to see a large body of work directed at experimenting with various kinds of TM implementation strategies, e.g. [1, 3–11, 8, 12–16]. What might be surprising is the little work devoted so far to the formalization of the *precise* guarantees that TM implementations should provide. Without such formalization, it is impossible to verify the correctness of these implementations, establish any optimality result, or determine whether various TM design trade-offs are indeed fundamental or simply artifacts of certain environments.

From a user's perspective, a TM should provide the same semantics as critical sections: transactions should appear as if they were executed sequentially, i.e., as if each transaction acquired a global lock for its entire duration. (Remember that the TM goal is to provide a simple abstraction to average programmers.) However, a TM implementation would be inefficient if it never allowed different transactions to run concurrently. Hence, we want to reason formally about executions with interleaving steps of arbitrary concurrent transactions. First, we need a way to state precisely whether a given execution in which a number of transactions execute steps in parallel "looks like" an execution in which these transactions proceed one after the other. That is, we need a *correctness condition* for TMs. Second, we should define when a TM implementation is allowed to *abort* a transaction that contends for shared data with concurrent transactions. Indeed, while the ability to abort transactions is essential for all optimistic schemes used by TMs, a TM that abuses this ability by aborting every transaction is, clearly, useless. Hence, we need to define *progress properties* of TM implementations.

We overview here our work on establishing the theoretical foundations of TMs [17–19]. We first present *opacity*—a correctness condition for TMs, which is indeed ensured by most TM implementations, e.g., DSTM [4], ASTM [5], SXM [20], JVSTM [6], TL2 [21], LSA-STM [11], RSTM [7], BartokSTM [8], McRT-STM [12], TinySTM [14], AVSTM [22], the STM in [23], and SwissTM [24]. We then define progress properties of the two main classes of existing TM implementations: *obstruction-free* [4] and *lock-based* ones. The intuition behind the progress semantics of such TMs has been known, but precise definitions were missing.

It is important to notice that the paper is only an overview of previously published results. In particular, we do not give here precise definitions of many terms that we use (and describe intuitively) or any proofs of the theorems that we state. Those definitions and proofs, as well as further details and discussions of the results presented here, can be found in [17–19].

2 Preliminaries

2.1 Shared Objects and their Implementations

Processes and objects. We consider a classical asynchronous shared-memory system [25, 26] of n processes $p_1, \ldots, p_n$ that communicate by executing operations on *(shared) objects*. An example of a very simple object is a *register*, which exports only *read* and *write* operations. Operation *read* returns the current state (value) of the register, and operation *write*(v) sets the state of the register to value v.

An execution of every operation is delimited by two *events*: the invocation of the operation and the response from the operation. We assume that, in every run of the system, all events can be totally ordered according to their execution time. If several events are executed at the same time (e.g., on multiprocessor systems),

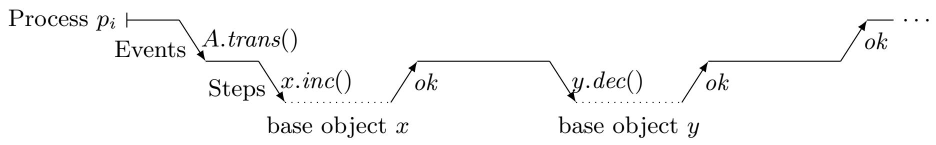

Fig. 1. An example execution of an operation *trans* on a shared object A by a process p_i. Operation *trans* is implemented using operations *inc* and *dec* on base objects x and y.

then they can be ordered arbitrarily. We call a pair of invocation of an operation and the subsequent response from this operation an *operation execution.*

An object x may be provided either directly in hardware, or *implemented* from other, possibly more primitive, *base* objects (cf. Figure 1). We call the events of operations on base objects *steps.* We assume that each process executes operations on shared objects, and on base objects, sequentially.

Wait-freedom. We focus on object implementations that are *wait-free* [25]. Intuitively, an implementation 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.

Computational equivalence. We say that object x *can implement* object y if there exists an algorithm that implements y using some number of instances of x (i.e., a number of base objects of the same type as x) and atomic (i.e., linearizable [27]) registers. We say that objects x and y are *equivalent* if x can implement y and y can implement x.

The power of a shared object. We use the *consensus number* [25] as a metric of the power of objects. The consensus number of an object x is the maximum number of processes among which one can solve (wait-free) *consensus* using any number of instances of x (i.e., base objects of the same type as x) and atomic registers.

The consensus problem consists for a number of processes to agree (*decide*) on a single value chosen from the set of values these processes have *proposed.* It is known that, in an asynchronous system, implementing wait-free consensus is impossible when only registers are available [28].

2.2 Transactional Memory (TM)

A TM enables processes to communicate by executing transactions. A transaction may perform operations on objects shared with other transactions, called *transactional* objects (or *t-objects*, for short), as well as local computations on objects inaccessible to other 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. We will call *t-variables* those t-objects that are registers, i.e., that provide only *read* and *write* operations.

Every transaction has a unique *identifier* (e.g., T_1, T_2, etc.). (We use the terms "transaction" and "transaction identifier" interchangeably.) Every transaction, upon its first action, is initially *live* and may eventually become either *committed* or *aborted*, as explained in the following paragraphs. A transaction that is not live does no longer perform any actions. Retrying an aborted transaction (i.e., the computation the transaction intends to perform) is considered in our model as a new transaction, with a different transaction identifier.

TM as a shared object. A TM can be viewed as an object with operations that allow for the following: (1) Executing any operation on a t-object x within a transaction T_k (returns the response of the operation or a special value A_k); (2) Requesting transaction T_k to be committed (operation $tryC(T_k)$ that returns either A_k or C_k); (3) Requesting transaction T_k to be aborted (operation $tryA(T_k)$ that always returns A_k). The special return value A_k (*abort* event) is returned by a TM to indicate that transaction T_k has been aborted. The return value C_k (*commit* event) is a confirmation that T_k has been committed.

As for other objects, we assume that every implementation of a TM is wait-free, i.e., that the individual operations of transactions are wait-free. This is indeed the case for most TM implementations (including lock-based ones; see [19]).

If x is a t-object (provided by a given TM), then we denote by $x.op_k \rightarrow v$ an execution (invocation and the subsequent response) of operation *op* on x by transaction T_k, returning value v. We also denote by A_k (and C_k) an abort (commit) event of transaction T_k.

Histories. Consider any TM and any run. A *history* (of the TM) is a sequence of invocation and response events of operations executed by processes on the TM in this run. Let M be any implementation of the TM. An *implementation history* of M is the sequence of (1) invocation and response events of operations executed by processes on M, and (2) the corresponding steps of M executed in a given run.

We say that transaction T_k is *committed* (respectively, *aborted*) in H, if H contains commit event C_k (resp., abort event A_k). A transaction that is neither committed nor aborted is called *live*. We say that transaction T_k is *forcefully aborted* in H, if T_k is aborted in H but there is no invocation of operation $tryA(T_k)$ in H. We say that T_k is *commit-pending* in H, if H contains an invocation of operation $tryC(T_k)$ but T_k is still live in H (i.e., operation $tryC(T_k)$ has not returned yet).

We say that a transaction T_k *precedes* a transaction T_m in history H, and write $T_k \prec_H T_m$, if T_k is committed or aborted and the last event of T_k precedes (in H) the first event of T_m. We say that transactions T_k and T_m are *concurrent* in a history H, if neither T_k precedes T_m, nor T_m precedes T_k (in H).

We say that history H is *sequential* if no two transactions in H are concurrent. We say that H is *complete* if H does not contain any live transaction.

We assume that every transaction T_k in H is executed by a single process. Conversely, we assume that every process p_i executes only one transaction at a time, i.e., that no two transactions are concurrent at any given process.

Sequential specification of a t-object. We use the concept of a *sequential specification* to describe the semantics of t-objects, as in [29, 27]. Intuitively, a sequential specification of a t-object x lists all sequences of operation executions on x that are considered correct when executed outside any transactional context, e.g., in a standard, single-threaded application.[1] For example, the sequential specification of a t-variable x, denoted by $\mathit{Seq}(x)$, is the set of all sequences of *read* and *write* operation executions on x, such that in each sequence that belongs to $\mathit{Seq}(x)$, every *read* (operation execution) returns the value given as an argument to the latest preceding *write* (regardless transaction identifiers). (In fact, $\mathit{Seq}(x)$ also contains sequences that end with a pending invocation of *read* or *write*, but this is a minor detail.) Such a set defines precisely the semantics of a t-variable in a single-threaded, non-transactional system.

3 Opacity

Opacity is a safety property that captures the intuitive requirements that (1) all operations performed by every *committed* transaction appear as if they happened at some single, indivisible point during the transaction lifetime, (2) no operation performed by any *aborted* transaction is ever visible to other transactions (including live ones), and (3) every transaction always observes a *consistent* state of the system.

To help understand the definition of opacity, we first consider very simple histories, and increase their complexity step by step. The precise definitions of the terms that correspond to the steps described here are given in [17].

Opacity is trivial to express and verify for sequential histories in which every transaction, except possibly the last one, is committed. Basically, if S is such a history, then S is considered correct, and called *legal*, if, for every t-object x, the subsequence S_x of all events in H executed on t-object x respects the semantics of x, i.e., S_x belongs to the sequential specification of x. For example, if a transaction T_i writes value v to a t-variable x at some point in history S, then all subsequent reads of x in S, performed by T_i or by a following transaction, until the next write of x, must return value v.

The situation becomes more difficult if S is sequential but contains some aborted transactions followed by committed ones. For example, if an aborted transaction T_i writes value v to a t-variable x (and no other transaction writes v to x), then only T_i can read v from x thereafter. A read operation on x executed by a transaction following T_i must return the last value written to x

[1] An operation execution specifies a transaction identifier, but the identifier can be treated as part of the arguments of the executed operation. In fact, in most cases, the semantics of an operation does not depend on the transaction that issues this operation.

by a preceding *committed* transaction. Basically, when considering a transaction T_i (committed or aborted) in S, we have to remove all aborted transactions that precede T_i in S. We then say that T_i is *legal* in S, if T_i together with all committed transactions preceding T_i in S form a legal history. Clearly, for an arbitrary sequential history S to be correct, all transactions in S must be legal.

To determine the opacity of an arbitrary history H, we ask whether H "looks like" some sequential history S that is correct (i.e., in which every transaction is legal). In the end, a user of a TM should not observe, or deal with, concurrency between transactions. More precisely, history S should contain the same transactions, performing the same operations, and receiving the same return values from those operations, as history H. We say then that H is *equivalent* to S. Equivalent histories differ only in the relative position of events of *different* transactions.

Moreover, the real-time order of transactions in history H should be preserved in S. That is, if a transaction T_i precedes a transaction T_k in H, then T_i must also precede T_k in S.

There is, however, one problem with finding a sequential history that is equivalent to a given history H: if two or more transactions are live in H, then there is no sequential history that is equivalent to H. Basically, if S is a sequential history, then $\prec_S$ must be a total order; however, if a transaction T_i precedes a transaction T_k in S, i.e., if $T_i \prec_S T_k$, then T_i must be committed or aborted. To solve the problem, observe that the changes performed by a transaction T_i should not become visible to other transactions until T_i *commits*. Transaction T_i commits at some point (not visible to the user) between the invocation and the response of operation $\mathit{tryC}(T_i) \to C_i$. That is, the semantics of T_i is the same as of an aborted transaction until T_i invokes $\mathit{tryC}(T_i)$, but this semantics might change (to the one of a committed transaction) at any point in time after T_i becomes commit-pending. Hence, we can safely transform an arbitrary history H into a *complete* history H' (called a *completion* of H) by (1) aborting all live and non-commit-pending transactions in H, and (2) committing or aborting every commit-pending transaction in H.

To summarize the above steps:

Definition 1. *A history H is* opaque *if there exists a sequential history S equivalent to any completion of H, such that (1) the real-time order of transactions in H is preserved in S, and (2) every transaction in S is legal in S.*

Note that the definition of opacity does not require every prefix of an opaque history to be also opaque. Thus, the set of all opaque histories is not prefix-closed. For example, while the following history is opaque:

$$H = \langle x.\mathit{write}(1)_1,\ x.\mathit{read}_2 \to 1,\ \mathit{tryC}(T_1) \to C_1,\ \mathit{tryC}(T_2) \to C_2\rangle,$$

the prefix $H' = \langle x.\mathit{write}(1)_1,\ x.\mathit{read}_2 \to 1\rangle$ of H is not opaque (assuming the initial value of x is 0), because, in H', transaction T_2 reads value written by T_1 that is not committed or commit-pending. However, a history of a TM is generated progressively and at each time the history of all events issued so far must be opaque. Hence, there is no need to enforce prefix-closeness in the definition of opacity, which should be as simple as possible.

The way we define the real-time ordering between transactions introduces a subtlety to the definition of opacity. Basically, the following situation is possible (and considered correct): a transaction T_1 updates some t-object x, and then some other transaction T_2 concurrent to T_1 observes an old state of x (from before the update of T_1) even after T_1 commits. For example, consider the following history (x and y are t-variables with initial value 0):

$$\begin{aligned} H = \langle & x.\mathit{read}_1 \to 0,\ x.\mathit{write}(5)_2,\ y.\mathit{write}(5)_2, \\ & \mathit{tryC}(T_2) \to C_2, y.\mathit{read}_3 \to 5,\ y.\mathit{read}_1 \to 0\rangle. \end{aligned}$$

In H, transaction T_1 appears to happen before T_2, because T_1 reads the initial values of t-variables x and y that are modified by T_2. Transaction T_3, on the other hand, appears to happen after T_2, because it reads the value of y written by T_2. Consider the following sequential history:

$$\begin{aligned} S = \langle & x.\mathit{read}_1 \to 0,\ y.\mathit{read}_1 \to 0,\ \mathit{tryC}(T_1) \to A_1, \\ & x.\mathit{write}(5)_2,\ y.\mathit{write}(5)_2,\ \mathit{tryC}(T_2) \to C_2, \\ & y.\mathit{read}_3(5),\ \mathit{tryC}(T_3) \to A_3\rangle. \end{aligned}$$

It is easy to see that S is equivalent to the completion $H \cdot \langle \mathit{tryC}(T_1) \to A_1, \mathit{tryC}(T_3) \to A_3\rangle$ of H, and that the real-time order of transactions in H is preserved in S. As, clearly, every transaction is legal in S, history H is opaque.

However, at first, it may seem wrong that the *read* operation of transaction T_3 returns the value written to y by the committed transaction T_2, while the following *read* operation, by transaction T_1, returns the old value of y. But if T_1 read value 5 from y, then opacity would be violated. This is because T_1 would observe an inconsistent state of the system: $x = 0$ and $y = 5$. Thus, letting T_1 read 0 from y is the only way to prevent T_1 from being aborted without violating opacity. Multi-version TMs, like JVSTM and LSA-STM, indeed use such optimizations to allow long read-only transactions to commit despite concurrent updates performed by other transactions. In general, it seems that forcing the order between operation executions of different transactions to be preserved, in addition to the real-time order of transactions themselves, would be too strong a requirement.

4 Obstruction-Free TMs

In this section, we define the class of obstruction-free TMs (OFTMs). We also determine the consensus number of OFTMs and show an inherent limitation of those TMs.

Our definition of an OFTM is based on the formal description of obstruction-free objects from [30]. In [18], we consider alternative definitions but we show, however, that these are computationally equivalent to the one we give here.

4.1 Definition of an OFTM

The definition we consider here uses the notion of *step contention* [30]: it says, intuitively, that a transaction T_k executed by a process p_i can be forcefully aborted only if some process other than p_i executed a step of the TM implementation concurrently to T_k.

More precisely, let E be any implementation history of any TM implementation M. We say that a transaction T_k executed by a process p_i encounters *step contention* in E, if there is a step of M executed by a process other than p_i in E after the first event of T_k and before the commit or abort event of T_k (if any).

Definition 2. *We say that a TM implementation M is* obstruction-free *(i.e., is an* OFTM*) if in every implementation history E of M, and for every transaction T_k in E, if T_k is forcefully aborted in E then T_k encounters step contention in E.*

4.2 The Power of an OFTM

We show that the consensus number of an OFTM is 2. We do so by first exhibiting an object, called *fo-consensus*, that is equivalent to any OFTM, and then showing that the consensus number of fo-consensus is 2. (The proofs of the theorems stated here are in [18].)

Intuitively, *fo-consensus* (introduced in [30] as “fail-only” consensus) provides an implementation of consensus (via an operation *propose*). That is, processes can use an fo-consensus object to agree on a single value chosen from the values that those processes propose, i.e., pass as a parameter to operation *propose.* However, unlike classical consensus, an fo-consensus object allows *propose* to *abort* when it cannot return a decision value because of concurrent invocations of *propose*. When *propose* aborts, it means that the operation did not take place, and so the value proposed using this operation has not been “registered” by the fo-consensus object (recall that only a value that has been proposed, and “registered”, can be decided). A process which *propose* operation has been aborted may retry the operation many times (possibly with different proposed value), until a decision value is returned. (For a precise definition of an fo-consensus object, see [18].)

Theorem 1. *An OFTM is equivalent to fo-consensus.*

Theorem 2. *Fo-consensus cannot implement (wait-free) consensus in a system of 3 or more processes.*

From Theorem 1, Theorem 2, and the claim of [30] that consensus can be implemented from fo-consensus and registers in a system of 2 processes, we have:

Theorem 3. *The consensus number of an OFTM is 2.*

Corollary 1. *There is no algorithm that implements an OFTM using only registers.*

4.3 An Inherent Limitation of OFTMs

We show that no OFTM can be strictly disjoint-access-parallel. To define the notion of strict disjoint-access-parallelism, we distinguish operations that modify the state of a base object, and those that are read-only. We say that two processes (or transactions executed by these processes) *conflict on a base object x*, if both processes execute each an operation on x and at least one of these operations modifies the state of x. Intuitively, a TM implementation M is *strictly disjoint-access-parallel* if it ensures that processes executing transactions which access disjoint sets of t-objects do not conflict on common base objects (used by M).

Theorem 4. *No OFTM is strictly disjoint-access-parallel.*

It is worth noting that the original notion of disjoint-access-parallelism, introduced in [31], allows for transactions that are *indirectly* connected via other transactions to conflict on common base objects. For example, if a transaction T_1 accesses a t-object x, T_2 accesses y, and T_3 accesses both x and y, then there is a dependency chain from T_1 to T_2 via T_3, even though the two transactions T_1 and T_2 use different t-objects. Disjoint-access-parallelism allows then the processes executing T_1 and T_2 to conflict on some base objects. Disjoint-access-parallelism in the sense of [31] can be ensured by an OFTM implementation, e.g., DSTM.

It is also straightforward to implement a TM that is strictly disjoint-access-parallel but not obstruction-free, e.g., using two-phase locking [32] or the TL algorithm [33].

5 Lock-Based TMs

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 [32].

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.

In this section, we define *strong progressiveness*—a progress property commonly ensured by lock-based TMs. We determine the consensus number of strongly progressive TMs, and show an inherent performance trade-off in those TMs. (The proofs of the theorems stated here can be found in [19].)

5.1 Strong Progressiveness

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, concurrent 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. (We assume here, for simplicity of presentation, that (1) all t-objects are t-variables, i.e., they export only *read* and *write* operations, and (2) there are no *false* conflicts. We discuss those assumptions in [19].)

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 multi-processor 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 [34]. 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.

Let H be any history, and T_i be any transaction in H. Intuitively, we denote by $CVar_H(T_i)$ the set of t-variables on which transaction T_i conflicts with any other transaction in history H.[2] If Q is any set of transactions in H, then $CVar_H(Q)$ denotes the union of sets $CVar_H(T_i)$ for all $T_i \in Q$, i.e., the set of t-variables on which any transaction in set Q conflicts with any other transaction in history H.

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.*

5.2 The Power of a Lock-Based TM

We show here that the consensus number of a strongly progressive TM is 2. First, we prove that a strongly progressive TM is computationally equivalent to a *strong try-lock* object that we describe in this section (and define precisely in [19]). That is, one can implement a strongly progressive TM from (a number

[2] For a precise definition, consult [19].

of) strong try-locks and registers, and vice versa. Second, we determine that the consensus number of a strong try-lock is 2.

All lock-based TMs we know of use (often implicitly) a special kind of locks, usually called *try-locks* [35]. 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.

Every try-lock L guarantees the following property, called *mutual exclusion*: no two processes hold L at the same time. 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 [36] properties of (blocking) locks.

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 [35]. It is trivial to transform a typical TAS or TATAS lock [36] into a strong try-lock [19].

Theorem 5. *A strongly progressive TM is equivalent to a strong try-lock.*

Theorem 6. *A strong try-lock has consensus number 2.*

Hence, by Theorem 5 and Theorem 6, the following theorem holds:

Theorem 7. *A strongly progressive TM has consensus number 2.*

Corollary 2. *There is no algorithm that implements a strongly progressive TM using only registers.*

5.3 Performance Trade-Off in Lock-Based TMs

We show that the space complexity of every strongly progressive TM that uses *invisible reads* is at least exponential with the number of t-variables available to transactions.[3] The invisible reads strategy is used by a majority of lock-based TM implementations [21, 7, 8, 12, 14, 24] 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

[3] In fact, the result holds also for TMs that ensure a property called *weak progressiveness*, which is strictly weaker than strong progressiveness [19].

invalidating each other's caches and causing high traffic on the inter-processor (or inter-core) bus. However, transactions that update t-variables do not know whether there are any concurrent transactions that read those variables. (For a precise definition of invisible reads, consult [19].)

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 8. *Every strongly progressive TM implementation that uses invisible reads and provides to transactions* N_{s} *t-variables of size* K_{s} *uses* $\Omega\left({K_{\mathrm{s}}}^{N_{\mathrm{s}}}/K_{\mathrm{b}}\right)$ *base objects of size* K_{b}*.*

This result 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 t-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 opacity. This requires (a) limiting the progress (strong progressiveness) of transactions when overflow occurs, and (b) preventing read-only transactions from being completely invisible [19]. 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.

6 Concluding Remarks

We gave an overview of our recent work on establishing the theoretical foundations of transactional memory (TM). We omitted many related results. We give here a short summary of some of those.

An important question is how to verify that a given history of a TM, or a given TM implementation, ensures opacity, obstruction-freedom, or strong progressiveness. In [17], we present a graph interpretation of opacity (similar in concept to the one of serializability [37, 38]). Basically, we show how to build a graph that represents the dependencies between transactions in a given history H. We then reduce the problem of checking whether H is opaque to the problem of checking the acyclicity of this graph. In [19], we provide a simple reduction scheme that facilitates proving strong progressiveness of a given TM implementation M. Roughly speaking, we prove that if it is possible to say which parts of the algorithm of M can be viewed as logical try-locks (in a precise sense we define in [19]), and if those logical try-locks are strong, then the TM is strongly progressive. In other words, if the locking mechanism used by M is based on (logical) strong try-locks, then M is strongly progressive.

The graph characterization of opacity and the reduction scheme for strong progressiveness do not address the problem of automatic model checking TM implementations. Basically, they do not deal with the issue of the unbounded number of states of a general TM implementation. In [39, 40], the problem is addressed for an interesting class of TMs. Basically, it is proved there that if a given TM implementation has certain symmetry properties, then it either violates opacity in some execution with only 2 processes and 2 t-variables, or ensures opacity in every execution (with any number of processes and t-variables). The theoretical framework presented in [39, 40] allows for automatic verifications of implementations such as DSTM or TL2 in a relatively short time. Work similar in scope is also presented in [41].

One of the problems that we did not cover is the semantics of memory transactions from a programming language perspective. A very simple (but also very convenient) interface to a TM is via an `atomic` keyword that marks those blocks of code that should be executed inside transactions. The possible interactions between transactions themselves are confined by opacity. However, opacity does not specify the semantics of the interactions between the various programming language constructs that are inside and outside atomic blocks. Some work on those issues is presented, e.g., in [42–46].

Acknowledgements

We thank Hagit Attiya, Aleksandar Dragojević, Pascal Felber, Christof Fetzer, Seth Gilbert, Vincent Gramoli, Tim Harris, Thomas Henzinger, Eshcar Hillel, Petr Kouznetsov, Leaf Petersen, Benjamin Pierce, Nir Shavit, Vasu Singh, and Jan Vitek for their helpful comments and discussions.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA. (1993)
2. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1992)
3. Shavit, N., Touitou, D.: Software transactional memory. In: PODC. (1995)
4. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC. (2003)
5. Maranthe, V.J., Scherer III, W.N., Scott, M.L.: Adaptive software transactional memory. In: DISC. (2005)
6. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. In: SCOOL. (2005)
7. Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W.N., Scott, M.L.: Lowering the overhead of software transactional memory. In: TRANSACT. (2006)
8. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: PLDI. (2006)
9. Spear, M.F., Marathe, V.J., N. Scherer III, W., Scott, M.L.: Conflict detection and validation strategies for software transactional memory. In: DISC. (2006)
10. Herlihy, M., Moir, M., Luchangco, V.: A flexible framework for implementing software transactional memory. In: OOPSLA. (2006)

11. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: DISC. (2006)
12. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI. (2006)
13. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in STM. In: PLDI. (2007)
14. Felber, P., Riegel, T., Fetzer, C.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP. (2008)
15. Gramoli, V., Harmanci, D., Felber, P.: Toward a theory of input acceptance for transactional memories. In: OPODIS. (2008)
16. Dragojević, A., Singh, A.V., Guerraoui, R., Singh, V.: Preventing versus curing: Avoiding conflicts in transactional memories. In: PODC. (2009)
17. Guerraoui, R., Kapałka, M.: On the correctness of transactional memory. In: PPoPP. (2008)
18. Guerraoui, R., Kapałka, M.: On obstruction-free transactions. In: SPAA. (2008)
19. Guerraoui, R., Kapałka, M.: The semantics of progress in lock-based transactional memory. In: POPL. (2009)
20. Herlihy, M.: SXM software transactional memory package for C# `http://www.cs.brown.edu/~mph`.
21. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: DISC. (2006)
22. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: DISC. (2008)
23. Raynal, M., Imbs, D.: An STM lock-based protocol that satisfies opacity and progressiveness. In: OPODIS. (2008)
24. Dragojević, A., Guerraoui, R., Kapałka, M.: Stretching transactional memory. In: PLDI. (2009)
25. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (January 1991) 124–149
26. Jayanti, P.: Robust wait-free hierarchies. Journal of the ACM **44**(4) (1997) 592–614
27. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (June 1990) 463–492
28. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(3) (April 1985) 374–382
29. Weihl, W.E.: Local atomicity properties: Modular concurrency control for abstract data types. ACM Transactions on Programming Languages and Systems **11**(2) (April 1989) 249–282
30. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: DISC. (2005)
31. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: PODC. (1994)
32. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11) (1976) 624–633
33. Dice, D., Shavit, N.: What really makes transactions fast? In: TRANSACT. (2006)
34. Blundell, C., Lewis, E.C., Martin, M.M.K.: Subtleties of transactional memory atomicity semantics. IEEE Computer Architecture Letters **5**(2) (2006)
35. Scott, M.L., Scherer III, W.N.: Scalable queue-based spin locks with timeout. In: PPoPP. (2001)

36. Raynal, M.: Algorithms for Mutual Exclusion. The MIT Press (1986)
37. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM **26**(4) (1979) 631–653
38. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems **8**(4) (1983) 465–483
39. Guerraoui, R., Henzinger, T., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI. (2008)
40. Guerraoui, R., Henzinger, T.A., Singh, V.: Completeness and nondeterminism in model checking transactional memorie. In: Concur. (2008)
41. O'Leary, J., Saha, B., Tuttle, M.R.: Model checking transactional memory with Spin. In: ICDCS. (2009)
42. Vitek, J., Jagannathan, S., Welc, A., Hosking, A.: A semantic framework for designer transactions. In: ESOP. (2004)
43. Jagannathan, S., Vitek, J., Welc, A., Hosking, A.: A transactional object calculus. Science of Computer Programming **57**(2) (2005) 164–186
44. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: POPL. (2008)
45. Moore, K.F., Grossman, D.: High-level small-step operational semantics for transactions. In: POPL. (2008)
46. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical weak-atomicity semantics for Java STM. In: SPAA. (2008)