THE THEORY OF TRANSACTIONAL MEMORY

Rachid Guerraoui

EPFL, Switzerland

rachid.guerraoui@epfl.ch

Michał Kapałka

EPFL, Switzerland

michal.kapalka@epfl.ch

Abstract

Transactional memory (TM) is a promising paradigm for concurrent programming. This paper is an overview of our recent work on defining a theory of TM. We first present a correctness condition of a TM, ensured by most existing TM implementations. Then, we describe two progress properties that characterize the two main classes of TM implementations: obstruction-free and lock-based TMs. We use these properties to establish several results on the inherent power and limitations of TMs.

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) [28] 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* [15]. 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. [28, 43, 26, 33, 8, 34, 23, 45, 27, 40, 23, 2, 44, 12, 14]. 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 some 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 [20, 19, 21]. We first present *opacity*—a correctness condition for TMs, which is indeed ensured by most TM implementations, e.g., DSTM [26], ASTM [33], SXM [24], JVSTM [8], TL2 [9], LSA-STM [40], RSTM [34], TinySTM [12], BartokSTM [23], McRT-STM [2], AVSTM [18], and the STM in [39]. The technical challenge in specifying opacity is the ability to reason about concurrent transactional executions in a general and high-level model (a) with arbitrary objects, beyond simple read/write variables, (b) supporting multiple versions of each object (i.e., multi-version protocols, used, e.g., in JVSTM), and (c) not precluding various TM strategies and optimization techniques, such as invisible reads, lazy updates, or caching.

At first glance, it seems very likely that such a criterion would correspond to one of the numerous ones defined in the literature, e.g., linearizability [29], serializability [37, 5], global atomicity [48], recoverability [22], or rigorous scheduling [7]. However, none of these criteria, nor any straightforward combination or extension thereof, is sufficient to describe the semantics of TM with its subtleties [20]. In particular, none of them captures exactly the very requirement that

every transaction, including a *live* (i.e., not yet completed) one, accesses a *consistent* state, i.e., a state produced by a sequence of previously committed transactions. A live transaction that accesses an inconsistent state might create significant dangers when executed within a general TM framework [45]. It is thus not surprising that most TM implementations employ mechanisms that disallow such situations, sometimes at a big cost. At a very high abstraction level, disallowing transactions to access inconsistent states resembles, in the database terminology, preventing *dirty reads* or, more generally, the *read skew* phenomenon [4], when generalized to all transactions (not only committed ones as in [4]) and arbitrary objects. Capturing this intuitive idea in a precise manner is not trivial.

It is worth noting that the difference between opacity and classical database properties like serializability is not "cosmetic". In fact, we showed in [20] that opacity is inherently more expensive to implement than the combination of (strict) serializability [37, 5] (or global atomicity [48]) and the strongest form of recoverability [22]. Basically, we proved a complexity lower bound that holds for TMs that ensure opacity, but does not hold for TMs that ensure serializability and recoverability.

We also present in this paper progress properties of the two main classes of existing TM implementations: *obstruction-free* [26] and *lock-based* ones. The intuition behind the progress semantics of such TMs has been known, but precise definitions were missing.

Roughly speaking, an obstruction-free TM (*OFTM*), such as DSTM, ASTM, RSTM, or SXM, guarantees progress for every thread of an application that executes transactions alone (i.e., without *contention*) for sufficiently long time. OFTMs are appealing in real-time systems, where priority inversion is an important issue, or within operating systems, where kernel-level transactions (e.g., inside interrupt handlers) must be able to preempt (and, in many cases, abort) user-level ones at any time [46]. In an OFTM, a transaction that is preempted, delayed, or even crashed cannot inhibit the progress of other transactions. This means that an OFTM cannot internally use any blocking mechanisms such as critical sections.

Many TM implementations that are considered effective, e.g., TL2, TinySTM, a version of RSTM, BartokSTM, or McRT-STM are, however, not obstruction-free. They internally use locking, in order to reduce the overheads of TM mechanisms. We define the progress semantics of lock-based TMs by introducing a property, which we call *strong progressiveness*,[1] and which stipulates the two following requirements: (1) A transaction that encounters no *conflict* must be able to commit (basically, a conflict occurs when two or more concurrent transactions

---

[1]We call it "strong" by opposition to a weaker form of progressiveness that we also introduce in [21].
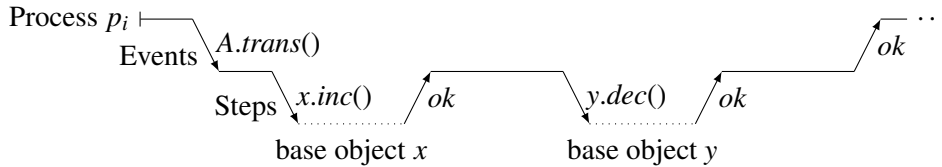
Figure 1: An example execution of an operation *trans* on a shared object *A* by a process p_i. Operation *trans* is implemented using operations *inc* and *dec* on base objects *x* and *y*.

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. Requirement (1) captures the common intuition about the progress of any TM (see [41]). Requirement (2) 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 [6]. Strong progressiveness is ensured by state of the art lock-based implementations, such as TL2, TinySTM, a version of RSTM, BartokSTM, and McRT-STM.[2]

We also give in this paper an overview of the theoretical results proved for OFTMs and strongly progressive (lock-based) TMs that ensure opacity.

It is important to notice that the paper is only an overview of previously published results. In particular, we do not give here any proofs of the theorems and lemmas that we state. Those proofs, as well as further details and discussions of the results presented here, can be found in [20, 19, 21].

2   Preliminaries

2.1   Shared Objects and their Implementations

Processes and objects. We consider a classical asynchronous shared-memory system [25, 32] 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*.

---

[2]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 [23] and [2], respectively.

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), 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*.

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

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 [29]) 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 [13]. Solving consensus consists in ensuring the following properties: (1) every value decided is one of the values proposed (*validity*); and (2) no two processes decide different values (*agreement*).

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 [21]).

If x is a t-object (provided by a given TM), then we denote by $inv_k(x.op)$, $ret_k(x.op \to v)$, and $x.op_k \to v$ an invocation, response, and execution (invocation and the subsequent response), respectively, 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 in a given run.

Let H be any (implementation) history of a TM and T_k be any transaction. We denote by $H|T_k$ the sequence of all events executed by T_k in H. We say that T_k is in H, and write $T_k \in H$, if there is some event executed by T_k in H, i.e., if $H|T_k$ is a non-empty sequence. We also denote by $H|x$ the restriction of H to events executed on t-object x.

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.

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 completed 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. Note also that, because a completed transaction does not perform any further action, a commit/abort event (if any) of every transaction T_k in H must be the last event in sub-history $H|T_k$.

Sequential specification of a t-object. We use the concept of a *sequential specification* to describe the semantics of t-objects, as in [48, 29]. 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.[3] For example, the sequential specification of a t-variable x, denoted by *Seq*(x), is the set of all sequences of *read* and *write* operation executions on x, such that in each sequence that belongs to *Seq*(x), every *read* (operation execution) returns the value given as an argument to the latest preceding *write* (regardless transaction identifiers). (In fact, *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.

More formally, let an *object-local history* of a t-object x be any prefix S of a sequence of operation executions, such that $S|x = S$. Then, a sequential specification *Seq*(x) of a t-object x can be any prefix-closed set of object-local histories of x. (A set Q of sequences is *prefix-closed* if, whenever a sequence S is in Q, every prefix of S is also in Q.)

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

---

[3]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.

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.

We start by explaining informally, step by step, how one can determine whether a given history of a TM ensures opacity. Next, we give a precise definition of opacity and the related terms, and provide an example.

3.1 Opacity Step by Step

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 Section 3.2.

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 sub-history $S|x$ 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 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 H should be preserved 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 transac-

tion 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 $tryC(T_i) \rightarrow C_i$. That is, the semantics of T_i is the same as of an aborted transaction until T_i invokes $tryC(T_i)$, 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' by (1) aborting all live and non-commit-pending transactions in H, and (2) committing or aborting every commit-pending transaction in H.

3.2 Definition of Opacity

Let S be any sequential history such that every transaction in S, except possibly the last one, is committed. We say that S is *legal* if, for every t-object x, the subsequence $S|x$ is in set $Seq(x)$ (the sequential specification of x).

Let S be any complete sequential history. We denote by $visible_S(T_i)$ the longest subsequence S' of S such that, for every transaction $T_k \in S'$, either (1) $k = i$, or (2) T_k is committed and $T_k \prec_S T_i$. We say that a transaction $T_i \in S$ is *legal in* S, if history $visible_S(T_i)$ is legal.

Let H and H' be any histories. We say that H and H' are *equivalent* if, for every transaction T_i, $H|T_i = H'|T_i$. We say that history H' *preserves the real-time order* of (equivalent) history H, if $\prec_H \subseteq \prec_{H'}$. That is, if $T_i \prec_H T_j$, then $T_i \prec_{H'} T_j$, for any two transactions T_i and T_j in H.

Intuitively, a *completion* of a history H is any history H' obtained from H by aborting or committing every commit-pending transaction in H, and by aborting every other live transaction in H. More precisely, a completion of a history H is any valid history H' of the form $H \cdot Q$, where Q is a sequence of invocations of operation $tryC$, commit events, and abort events, such that (1) every transaction that is live and not commit-pending in H is aborted in H', and (2) every transaction that is commit-pending in H is either committed or aborted in H'. In particular, the only completion of a complete history H is H itself.

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

$$H = \langle x.write(1)_1, \ x.read_2 \rightarrow 1, \ tryC(T_1) \rightarrow C_1, \ tryC(T_2) \rightarrow C_2 \rangle,$$

the prefix $H' = \langle x.write(1)_1, \ x.read_2 \rightarrow 1 \rangle$ 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):

$$
\begin{aligned}
H \ = \ & \langle x.read_1 \rightarrow 0, \ x.write(5)_2, \ y.write(5)_2, \\
& \quad tryC(T_2) \rightarrow C_2, y.read_3 \rightarrow 5, \ y.read_1 \rightarrow 0 \rangle.
\end{aligned}
$$

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:

$$
\begin{aligned}
S \ = \ & \langle x.read_1 \rightarrow 0, \ y.read_1 \rightarrow 0, \ tryC(T_1) \rightarrow A_1, \\
& \quad x.write(5)_2, \ y.write(5)_2, \ tryC(T_2) \rightarrow C_2, \\
& \quad y.read_3(5), \ tryC(T_3) \rightarrow A_3 \rangle.
\end{aligned}
$$

It is easy to see that S is equivalent to the completion $H \cdot \langle tryC(T_1) \rightarrow A_1, tryC(T_3) \rightarrow A_3 \rangle$ of H, and that S preserves the real-time order of H. 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.
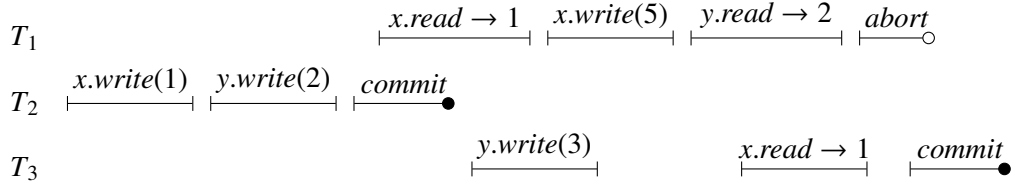
Figure 2: An opaque history H

3.3 Example of an Opaque History

To illustrate our definition, consider the following history H, of three transactions accessing two t-variables (x and y), corresponding to the execution depicted in Figure 2:

$$
\begin{aligned}
H = \;& \langle x.write(1)_2,\; y.write(2)_2,\; inv(tryC(T_2)),\; inv_1(x.read),\; C_2, \\
& inv_3(y.write(3)),\; ret_1(x.read \to 1),\; inv_1(x.write(5)),\; ret_3(y.write(3)), \\
& ret_1(x.write(5)),\; inv_1(y.read),\; inv_3(x.read),\; ret_1(y.read \to 2), \\
& inv(tryC(T_1)),\; ret_3(x.read \to 1),\; inv(tryC(T_3)),\; A_1,\; C_3 \rangle.
\end{aligned}
$$

Clearly, the only completion of H is H itself as there is no live transaction in H. Moreover, $\prec_H = \{(T_2, T_3)\}$ because T_1 is concurrent with T_2 and T_3. Therefore, we can find three sequential histories that are equivalent to H and preserve the real-time order of H (relation $\prec_H$). However, T_1 reads from x the value that has been written by committed transaction T_2. Thus, a sequential history in which T_1 precedes T_2 is not legal. Similarly, T_3 cannot precede T_1: T_1 reads from y the value written by T_2 and not the value written by the committed transaction T_3. Consider the following sequential history $S = H|T_2 \cdot H|T_1 \cdot H|T_3$. Clearly, S is equivalent to H and preserves the real-time order of H. Furthermore, every transaction is legal in S, because sequential histories $visible_H(T_2) = H|T_2$, $visible_H(T_1) = H|T_2 \cdot H|T_1$, and $visible_H(T_3) = H|T_2 \cdot H|T_3$ are all legal. Therefore, history H is opaque.

4 Obstruction-Free TMs

In this section, we define precisely 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 [3]. In [19], 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* [3]: 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.

Intuitively, *fo-consensus* (introduced in [3] as "fail-only" consensus) provides an implementation of consensus (via an operation *propose*), but 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.

More precisely, let D be any set, such that $\perp \notin D$. Fo-consensus (object) implements a single operation, called *propose*, that takes a value $v \in D$ as an argument and returns a value $v' \in D \cup \{\perp\}$. If a process p_i is returned a non-$\perp$ value v' from *propose*(v), we say that p_i *decides* value v'. Once p_i decides some value, p_i does not invoke *propose* anymore. When operation *propose* returns $\perp$, we say that the operation *aborts*.

Consider any implementation I of a fo-consensus object. We say that an execution of operation *propose* of I by a process p_i is *step contention-free* if there is no step of I executed by a process other than p_i between the invocation and the response events of this operation execution. Fo-consensus satisfies the following properties: (1) if some process decides value v, then v is proposed by some *propose* operation that does not abort; (2) no two processes decide different values;

and (3) if a *propose* operation is step contention-free, then the operation does not abort.

Theorem 3. *An OFTM is equivalent to fo-consensus.*

Theorem 4. *Fo-consensus cannot implement (wait-free) consensus in a system of 3 or more processes.*

From Theorem 3, Theorem 4, and the claim of [3] that consensus can be implemented from fo-consensus and registers in a system of 2 processes, we have:

Theorem 5. *The consensus number of an OFTM is 2.*

Corollary 6. *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). More precisely:

Definition 7. *We say that a TM implementation M is* strictly disjoint-access-parallel *if, for every implementation history E of M and every two transactions T_i and T_k in E, if T_i and T_k conflict on some base object, then T_i and T_k both access some common t-object.*

Theorem 8. *No OFTM is strictly disjoint-access-parallel.*

It is worth noting that the original notion of disjoint-access-parallelism, introduced in [30], 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 [30] 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 [11] or the TL algorithm [10].

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 [11].

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.

For simplicity of presentation, we assume in this section that all t-objects are t-variables. That is, they export only *read* and *write* operations. We discuss how to deal with arbitrary t-objects in Section 5.4.

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.[4]

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 [6]. Finally,

---

[4]We assume no *false* conflicts here; we discuss this assumption in Section 5.4.

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 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 9. *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 define in this section. 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.

All lock-based TMs we know of use (often implicitly) a special kind of locks, usually called *try-locks* [42]. 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) *try-lock*, 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.

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 [38] properties of (blocking) locks.

Definition 10. *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 [42]. It is trivial to transform a typical TAS or TATAS lock [38] into a strong try-lock [21].

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

Theorem 12. *A strong try-lock has consensus number 2.*

Hence, by Theorem 11 and Theorem 12, the following theorem holds:

Theorem 13. *A strongly progressive TM has consensus number 2.*

Corollary 14. *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.[5] The invisible reads strategy is used by a majority of lock-based TM implementations [9, 34, 23, 2, 12] 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. (For a precise definition of invisible reads, consult [21].)

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 15. *Every strongly progressive TM implementation that uses invisible reads and provides to transactions N_s t-variables of size K_s uses $\Omega\left(K_s^{N_s}/K_b\right)$ 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 [21]. 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.

5.4   Generalizing Strong Progressiveness

The two major assumptions we made when defining the notion of strong progressiveness were that all t-objects are t-variables (i.e., they support only *read* and *write* operations), and (implicitly) 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 definitions).

Arbitrary t-objects. Object-based TMs support t-objects of arbitrary type. However, most of them classify all the operations of t-objects as either read-only or update ones. In those cases, there is no need to consider arbitrary t-objects, because

---

[5]In fact, the result holds also for TMs that ensure a property called *weak progressiveness*, which is strictly weaker than strong progressiveness [21].

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-objects of any type. In this case, one can redefine the notion of a conflict to account for arbitrary t-objects. 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 redefined, our definitions of progress properties remain correct even for t-objects with arbitrary operations. If we assume that a TM must support t-variables (in addition to other t-objects), then also the consensus number and complexity lower bound results hold for those TMs.

False conflicts. Many lock-based TMs employ a hash function to map a t-variable (or, in general, a t-object) 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*.) 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 any property that relies on the notion of a conflict) would no longer be visible, and very meaningful, to a user.

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 [20], we present a graph interpretation of opacity (similar in concept to the one of serializability [37, 5]). 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 [21], 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 [21]), 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 [16, 17], 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 [16, 17] allows for automatic verifications of implementations such as DSTM or TL2 in a relatively short time.

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 [47, 31, 1, 36, 35].

Acknowledgements

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *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 *PLDI*, 2006.

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

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

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

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

[7] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, 1991.

[8] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In *SCOOL*, 2005.

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

[10] D. Dice and N. Shavit. What really makes transactions fast? In *TRANSACT*, 2006.

[11] 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.

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

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, Apr. 1985.

[14] V. Gramoli, D. Harmanci, and P. Felber. Toward a theory of input acceptance for transactional memories. In *OPODIS*, 2008.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[16] R. Guerraoui, T. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *PLDI*, 2008.

[17] R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memorie. In *Concur*, 2008.

[18] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, 2008.

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

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

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

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

[23] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, 2006.

[24] M. Herlihy. SXM software transactional memory package for C#. `http://www.cs.brown.edu/~mph`.

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

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

[27] M. Herlihy, M. Moir, and V. Luchangco. A flexible framework for implementing software transactional memory. In *OOPSLA*, 2006.

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

[29] 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.

[30] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, 1994.

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

[32] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[33] V. J. Maranthe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, 2005.

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

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

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

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

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

[39] M. Raynal and D. Imbs. An STM lock-based protocol that satisfies opacity and progressiveness. In *OPODIS*, 2008.

[40] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.

[41] M. L. Scott. Sequential specification of transactional memory semantics. In *TRANS-ACT*, 2006.

[42] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. In *PPoPP*, 2001.

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

[44] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI*, 2007.

[45] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC*, 2006.

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

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

[48] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, Apr. 1989.