On Obstruction-Free Transactions

Rachid Guerraoui
School of Computer and Communication Sciences, EPFL
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Michał Kapałka
School of Computer and Communication Sciences, EPFL
Lausanne, Switzerland
michal.kapalka@epfl.ch

ABSTRACT

This paper studies obstruction-free software transactional memory systems (OFTMs). These systems are appealing, for they combine the atomicity property of transactions with a liveness property that ensures the commitment of every transaction that eventually encounters no contention.

We precisely define OFTMs and establish two of their fundamental properties. First, we prove that the consensus number of such systems is 2. This indicates that OFTMs cannot be implemented with plain read/write shared memory, on the one hand, but, on the other hand, do not require powerful universal objects, such as compare-and-swap. Second, we prove that OFTMs cannot ensure disjoint-access-parallelism (in a strict sense). This may result in artificial "hot spots" and thus limit the performance of OFTMs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Theory, Algorithms

Keywords

Transactional memory, obstruction-freedom, consensus number, impossibility

1. INTRODUCTION

Transactional memory (TM) is a new software paradigm in which processes (threads) of an application communicate using lightweight, in-memory transactions. Basically, a process that wants to access a shared data structure executes some operations on this structure inside an atomic program called a *transaction*. When the transaction commits, all these operations appear as if they took place instantaneously, at some single, unique point in time. When the transaction aborts, however, all the operations are rolled back and their effects are never visible to other transactions. This method of providing thread-safety is as easy to use as coarse-grained locking and, in many cases, nearly as efficient on multi-core systems as hand-crafted, fine-grained locking [21, 25]. Moreover, unlike lock-based schemes, transactions are composable [17].

Transactional memory can be implemented as a software library. Such a TM implementation is called a *software TM (STM)* [29]. A specific class of STMs is particularly interesting: those called *obstruction-free* STMs [19] (which we call *OFTMs*). Roughly speaking, an OFTM guarantees progress for every process that eventually does not encounter contention. OFTMs are appealing in real-time systems where priority inversion is an important issue, as well as 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 [30]. In an OFTM, a process that is preempted, delayed or even crashed cannot inhibit the progress of other processes.

Whereas a lot of practical experiments have been conducted to fine tune the performance of OFTMs [19, 26, 1, 8, 30], very little research has been devoted to establish the theoretical power and limitations of such systems. This paper is a preliminary step in that direction.

A typical OFTM. All current OFTMs [19, 26, 1, 8, 30] employ the same basic high-level principle, and differ mostly in the optimization techniques they use to lower the overhead of transaction processing. The best way to explain the principle is to look at the first, and arguably simplest, OFTM called DSTM [19].

The basic idea is the following. To update some object x, a transaction T_i acquires an exclusive ownership of x (using a *compare-and-swap* (CAS) operation). From this moment on, x contains the information that it is owned by T_i and points to the *transaction descriptor* of T_i, which indicates whether T_i is still live, already committed or aborted. The ownership of x by T_i is exclusive but revocable: otherwise the STM would not be obstruction-free. Indeed, if another transaction T_k wants to update x before T_i is completed, T_k cannot get blocked waiting for T_i to terminate. A contention manager might tell T_k to back off for some fixed time (maybe random) to give T_i a chance, but eventually T_k must be able to abort T_i and acquire x without any interaction with T_i.

If T_i wants to read some object y, then T_i just needs to make sure that no other transaction T_k is currently updating y; if not, then T_i may have to eventually abort T_k. Once y is not updated by any transaction, T_i simply reads the current state of y, without writing anything to shared memory. Later, when T_i reads other objects, or tries to commit, the state of y is re-read to ensure that T_i still observes a consistent state of the system (i.e., that nobody changed y after it was read by T_i).

Once a transaction T_i acquires ownership of all the objects T_i wants to update (and reads all objects it had to), T_i tries to commit by atomically changing its status field from "live" to "committed" (using CAS). Clearly, T_i will fail to do so if any other transaction has already aborted T_i, by atomically changing the status field of T_i from "live" to "aborted" (again, using CAS). Once T_i commits, all further transactions see the updates done by T_i.

The computational power of an OFTM. DSTM uses CAS for both object acquisition and transaction commitment. In fact, all current OFTMs use CAS, which seems at first glance necessary to ensure both obstruction-freedom and atomicity. It is natural to ask whether we can implement an OFTM using objects that support only weaker operations than CAS (i.e., objects lower in the Herlihy's hierarchy [18]), e.g., read-write registers.

An object that supports a CAS operation (e.g., a CAS object) is *universal*. It can wait-free [18] implement any atomic object shared by any number of processes. On the contrary, an OFTM seems generally unable to implement wait-free atomic objects, for it can abort any transaction when some other transaction is concurrently executing steps. This suggests that OFTMs have lower computational power than CAS, and might be implemented using weaker objects.

We show in Section 4 that an OFTM is not universal for 3 or more processes. The proof goes through showing a computational equivalence of an OFTM to "fail-only" consensus, an object introduced in [6] and called here *fo-consensus*. This equivalence result is, we believe, interesting in its own right, for it may help devising further impossibilities (as fo-consensus has much simpler semantics than an OFTM). We prove here that fo-consensus cannot solve (wait-free) consensus for 3 processes or more and, using the observation of [6] (that fo-consensus can implement consensus in a system of 2 processes), we establish that the *consensus number* of an OFTM is 2. This means that, on the one hand, an OFTM cannot be implemented from only read-write registers, but, on the other hand, objects as powerful as CAS are not necessary to implement an OFTM. In fact, we exhibit an OFTM implementation that uses only one-shot objects of consensus number 2 and registers.

The parallelism of an OFTM. An STM implementation should minimize the interactions between transactions that access disjoint sets of (application-level) objects. Basically, if a transaction T_i does not access any object accessed by another transaction T_k, then neither of these transactions should delay the other one. Ideally, the STM should ensure that the processes executing T_i and T_k do not perform conflicting operations on the underlying memory locations. This property prevents artificial "hot spots"—memory locations that are accessed concurrently and in a conflicting way by unrelated transactions. These may provoke "useless" cache invalidations—thus decreasing performance. We call this property *strict disjoint-access-parallelism*[1].

Lock-based TM implementations, most of which use some variant of the known two-phase locking protocol, are usually strictly disjoint-access-parallel (e.g., TL [11]). Notable exceptions are those TMs that use global timestamps in order to speed up the read validation process, e.g., TL2 [10] and TinySTM [13]. In those implementations, every transaction has to access a common memory location to determine its timestamp.

It could seem, at first, that DSTM (and other OFTMs) is strictly disjoint-access-parallel. Unfortunately, this is not the case. Consider a transaction T_m that updated both x and y, and then got suspended for a long time. Objects x and y both point to the transaction descriptor of T_m. Thus, a transaction T_i when accessing x, and a transaction T_k when accessing y will both go to T_m's transaction descriptor and possibly update it in order to abort T_m. Hence, T_i and T_k may contend on the same memory location, even if T_i and T_k use only object x and y, respectively.

Unfortunately, there is no remedy to this situation: If a separate transaction descriptor of T_m is created for each object, then there is no way to atomically commit T_m. Indeed, if the status of T_m is changed in the descriptor pointed by x, and not yet by y, then some transactions may read the values written by T_m and commit, thus forcing T_m to also eventually commit, while the others may read old object values and cause an irrecoverable conflict with T_m, thus requiring that T_m is eventually aborted.

In fact, we prove in this paper (Section 5) that no OFTM can be strictly disjoint-access-parallel. This means that transactions that are themselves unrelated, but happen to have some indirect connection via other transactions, can delay each other.

Scoping the Results. Proving our results requires a precise definition of the notion of an OFTM. While indeed the term has been widely used, it has never been formally stated. We propose a precise, yet general, definition of an OFTM (Section 2) and we prove its equivalence to two alternatives (Section 3).

For presentation simplicity, we consider, as a safety property of an OFTM, basic serializability [27]. Our results also hold for OFTMs that ensure the stronger opacity property [16], which preserves real-time ordering and ensures that non-committed transactions observe a consistent state of the system. The results also hold for a weak definition of an OFTM that allows crashed processes to block the progress of others even for very a long, but always finite, period of time [9, 4] (see Section 6).

2. PRELIMINARIES

2.1 Overview

Processes. We consider a classical asynchronous shared-memory system [18, 24] of n processes (threads) $p_1, \ldots, p_n$, of which $n-1$ may, at any time, fail by *crashing*. Once a process crashes, it does not take any further actions. The failures model the fact that processes may often be delayed arbitrarily (e.g., when de-scheduled, waiting for IO operations,

[1]Among the properties defined in [3], strict disjoint-access-parallelism corresponds to 1-local contention (or 0-local contention according to [7]). Our property also expresses similar goals as the notion of disjoint-access-parallelism introduced in [23]. However, the property of [23], unlike our strict disjoint-access-parallelism, allows transactions that are *indirectly* connected (via other transactions), to delay each other.

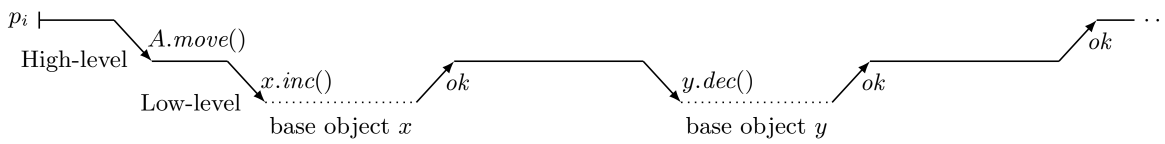

Figure 1: An example execution of an operation *move* on a high-level object A by a process p_i. Operation *move* is implemented using operations *inc* and *dec* on base objects x and y.

or encountering a page fault), in which case they should not block other processes (the very idea behind obstruction-freedom). A process that does not crash (in a given execution) is said to be *correct*.

Objects. We consider the actions taken by processes at two levels (cf. Figure 1). At the low-level, we consider processes executing operations on *base objects* (e.g., hardware memory locations). At a high level, we consider (the same) processes executing operations on *high-level objects* that are implemented using base objects. When a process p_i invokes an operation *op* on a high-level object x, p_i follows the implementation of *op* that determines the operations on base objects p_i must execute in order to provide the correct semantics of *op* on x. The two-level distinction is relative: an object x is a high-level object when we look at its implementation, or a base object when we look at another high-level object y implemented from x (and possibly other base objects).

An execution of each operation is delimited by two *events*: the invocation and the response from the operation. We assume that, in every execution, 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), they can be ordered arbitrarily. Events of operations on high-level objects, issued by a process p_i, are local to p_i. However, p_i's events on base objects, which we call *steps*, can be visible to other processes. We assume that every shared object[2] is *wait-free*: if a correct process p_i invokes an operation on x, then p_i eventually returns from the operation.

A *register* object exports only operations: *read* that returns the current value (state) of the register, and *write*(v) that changes the state of the register to value v. Thus, a register acts as a simple variable, and so in the algorithms we use registers as variables instead of specifying explicitly the *read* and *write* operations. We assume that every register is *atomic* (i.e., *linearizable* [22]).

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 objects of the same type as x) and registers. We say that objects x and y are *equivalent* if x can implement y and y can implement x.

Histories. A *(high-level) history* of a shared object x is a sequence of all events of operations executed on x by all processes in a given execution. A *low-level history* of an implementation I_x of a high-level object x is a sequence of: (1) all events of operations executed on x, and (2) all steps executed on behalf of I_x, by all processes in a given execution. We assume a typical well-formedness property of every (high-level or low-level) history: at each process p_i, no two operations on high-level objects (and no two operations on base objects) overlap, i.e., p_i executes operations on high-level objects, and also on base objects, sequentially, as it is shown in Figure 1 (for a precise definition of a well-formed history, see the full version of this paper [15]).

[2]When we say "(shared) object x" we mean "base or high-level object x".

2.2 Transactional Memory

Overview. A transactional memory (TM) allows for processes to communicate by reading or updating, within *transactions*, shared variables, which we call here *transactional variables* (or *t-variables*, for short)[3]. Once a transaction T_k executed by a process p_i *commits*, all the changes to t-variables done by p_i within T_k are atomically applied. If T_k *aborts*, however, the changes are rolled back and are never visible to other transactions.

Every transaction has a unique *transaction identifier* (e.g., T_k, $T_{i,k}$, etc.). A transaction T_k is executed, in a given low-level history E, by at most one process, denoted by $p_E(T_k)$[4]. We assume that once T_k is committed or aborted, no process performs any operations within T_k. Thus, when a process p_i wants to restart a computation of a transaction that has just (become) aborted, p_i simply repeats the computation within a new transaction (with a different identifier).

TM as a shared object. A TM can be viewed as an object with operations that allow for the following: (1) reading or writing a t-variable 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), and (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. For simplicity, we say that a transaction T_k performs a TM operation, or executes an event or step, meaning that some process p_i performs the operation, or executes the event or step of the considered STM implementation, within T_k.

It is worth noting that the TM operations described here are used only on the interface between an application (transactions) and a TM. When processes execute steps of a TM implementation itself, they may do much more than the TM external interface allows for. For example, they may abort transactions executed by other processes, or even help other

[3]In general, transactions may use objects of any type; however, the proofs of our results are more easily explained with only read-write t-variables (transactional registers). This does not, however, limit the generality of our results, as explained in Section 6.

[4]We use unique transaction identifiers for convenience and simplicity of notation. Such identifiers can be generated locally by each process, e.g., by combining the id of the process with the value of a process-local transaction counter.

processes in processing their transactions[5]. (Note that the same processes execute transactions on behalf of both an application and a TM implementation if the TM is not provided by hardware.)

Transactions. Let H be a (low-level or high-level) history of a TM (shared object) and T_k be a transaction. 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.

We say that a 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 committed or aborted (in H) is *completed*. A transaction that is not completed (in H) is called *live*. We say that a transaction T_k is *forcefully aborted* in H, if T_k is aborted in H but T_k has not issued $tryA(T_k)$ in H. (The ability to forcefully abort a transaction is essential for optimistic concurrency schemes.)

We say that a transaction T_k *precedes* a transaction T_m (in a history H), 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 assume that transactions at any single process are never concurrent.

Serializability. Serializability [27] is a safety property that describes the semantics of a TM. Intuitively, serializability requires that in every history H of a TM, all transactions that *have committed* in H issue the same invocation events and receive the same responses as in some *sequential* history S consisting of those transactions (in a sequential history, no two transactions are concurrent). A transaction T_k commits somewhere between its invocation of operation $tryC(T_k)$ and the subsequent C_k response. Thus, a transaction that is *commit-pending*, i.e., that has invoked $tryC(T_k)$, but has not received a matching response yet, may have already committed (or not). (We recall the precise definition of serializability, as used in our correctness proofs, in [15]).

2.3 Obstruction-Free STM Implementations

In this section, we define precisely what an OFTM is. We give here a definition based on the formal description of obstruction-free objects from [6]. We use this OFTM definition throughout our paper. Later, in Section 3, we consider alternative definitions. We show, however, that these are computationally equivalent to the one we give here (Section 3), and that the results proved in this paper hold also for those definitions (Section 6).

The definition we consider here uses the notion of *step contention* [6]: 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 concurrently to T_k.

More precisely, let E be any low-level history of some STM implementation I. We say that a transaction T_k encounters *step contention* in E, if there is a step of a process other than $p_E(T_k)$ in E after the first event of T_k and before the commit or abort event of T_k (if any).

Definition 1. *We say that an STM implementation I is* obstruction-free *(i.e., is an* OFTM*) if in every low-level history E of I, and for every transaction $T_k \in E$, if T_k is forcefully aborted in E, then T_k encounters step contention in E.*

[5]The TM model given here also does not support non-transactional accesses to t-variables, which are outside the scope of this paper.

3. ALTERNATIVE DEFINITIONS OF OFTM

Alternative definitions of OFTMs based on the concept of *interval contention* (instead of step contention) can also be considered [4]. Basically, we can allow a transaction T_k to be forcefully aborted only when there is a transaction T_i that is concurrent to T_k and that is executed by a process that has not crashed yet. We have at least two possible definitions here: In the simplest case (which we call *ic-obstruction-freedom*), we can assume that a process that crashes cannot cause any further transaction to be forcefully aborted. A weaker variant of this definition (*eventual ic-obstruction-freedom*), inspired by [4], allows a crashed process to obstruct other processes (and their transactions) for arbitrary, but finite time. More specifically:

Definition 2. *We say that an STM implementation I is* ic-obstruction-free *(i.e., is an* ic-OFTM*), if in every low-level history E of I, and for every transaction $T_k \in E$, if T_k is forcefully aborted, then there exists a transaction T_i concurrent to T_k, such that process $p_E(T_i)$ has not crashed before the first event of T_k.*

Definition 3. *We say that an STM implementation I is* eventually ic-obstruction-free *(i.e., is an* eventual ic-OFTM*), if for every low-level history E of I there exists a finite period of time d, such that for every transaction $T_k \in E$ that is forcefully aborted, there exists a transaction T_i concurrent to T_k, such that process $p_E(T_i)$ has not crashed earlier than d before the first event of T_k.*

Clearly, every STM that is obstruction-free is also ic-obstruction-free: a process that has crashed can no longer perform any steps. The opposite is also true: because slow processes cannot be distinguished from crashed ones, the only way for a process p_i to ensure that other processes are alive is for p_i to observe steps of other processes. Thus:

Theorem 4. *Every OFTM is an ic-OFTM, and every ic-OFTM is an OFTM.*

Clearly, every OFTM that is (ic-)obstruction-free is also eventually ic-obstruction-free. However, the opposite is not true: a history of an eventual ic-OFTM may contain finite sequences of forcefully-aborted transactions that are concurrent only to some transaction executed by a crashed process.

Nevertheless, one can *implement* an (ic-)OFTM using an eventual ic-OFTM. The transformation is not straightforward, though. For example, one could think that simply restarting every forcefully aborted transaction several times would provide ic-obstruction-freedom. But an eventual ic-OFTM may forcefully abort transactions at a single process arbitrarily (albeit finitely) many times in a row with ic-obstruction-freedom violation. Furthermore, restarting a computation of a transaction cannot be done by a TM implementation itself: the restarted transaction may see different states of the system and it is up to the application using a TM to decide then what operations on which t-variables to perform within the transaction.

Due to space limitations, we show the implementation of an ic-OFTM using an eventual ic-OFTM in the full version of this paper [15]. Hence, we prove the following result:

Theorem 5. *Every eventual ic-OFTM can implement an OFTM. Every OFTM is an eventual ic-OFTM.*

4. AN OFTM CANNOT SOLVE 3-CONSENSUS

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 in which some processes may crash, solving consensus is impossible when only registers are available [14].

In this section, we show that it is impossible to solve consensus for 3 processes (called *3-consensus*) using only OFTMs and registers (as base objects). We prove this result in two steps: First, we show that an OFTM is equivalent to a "fail-only" consensus object [6] (or fo-consensus, for short), i.e., that an OFTM can implement fo-consensus and vice versa. Then, we prove that fo-consensus cannot implement 3-consensus.

4.1 Definitions

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). The *consensus number* of an object O is the maximum number of processes among which one can solve consensus using any number of instances of O (i.e., base objects of the same type as O) and registers.

Intuitively, *fo-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 $\bot \notin D$. Fo-consensus (object) implements a single operation, called *propose*, that takes a value $v \in D$ as an argument and returns a value $v' \in D \cup \{\bot\}$. If a process p_i is returned a non-$\bot$ 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 $\bot$, we say that the operation *aborts*.

Let E be any low-level history of a fo-consensus implementation I_c. We say that a *propose* operation executed by a process p_i is *step contention-free* (in E) if there is no step of a process other than p_i between the invocation and the response events of this operation (in E). Fo-consensus satisfies the following properties (for every E): (1) *fo-validity* says that if some process decides value v, then v is proposed by some *propose* operation that does not abort; (2) *agreement* says that no two processes decide different values; and (3) *fo-obstruction-freedom* says that if a *propose* operation is step contention-free, then the operation does not abort.

4.2 Equivalence

We prove that an OFTM is equivalent to fo-consensus by showing that: (1) one can implement fo-consensus using an OFTM base object, and (2) one can implement an OFTM using fo-consensus objects and registers.

Lemma 6. *Every OFTM can implement fo-consensus.*

Algorithm 1: Implementing fo-consensus from an OFTM (code for a process p_i)

```
uses: V – a t-variable
initially: V = ⊥, k = 0
upon propose(v_i) do
    k ← k + 1;
    within transaction T_{i,k} do
        if V = ⊥ then V ← v_i;
        else v_i ← V;
    on event C_{i,k} do return v_i;
    on event A_{i,k} do return ⊥;
```

Proof (sketch). Implementing fo-consensus using an OFTM is straightforward. Algorithm 1 does so by having every process p_i that invokes *propose* use a transaction $T_{i,k}$[6] to atomically change the value of t-variable V from $\bot$ to the value proposed by p_i. If $T_{i,k}$ commits, then p_i can safely decide on the non-$\bot$ value that is in V (written by $T_{i,k}$ or read by $T_{i,k}$). Indeed, by serializability, only one committed transaction can observe that $V = \bot$ and set V to a non-$\bot$ value. Thus, agreement and fo-validity are ensured. Furthermore, $T_{i,k}$ can be aborted only if $T_{i,k}$ encounters step contention. But then the containing *propose* operation is not step contention-free and can abort without violating fo-obstruction-freedom. □

For simplicity, we use the "within transaction T_m … on event …" notation in Algorithm 1 instead of referring explicitly to the TM operations described in Section 2.2. The precise meaning of this notation is the following: A read (or write) of a t-variable x inside a "within transaction T_m … on event" block B means that transaction T_m (i.e., the process p_i that executes T_m) should invoke a read (write) operation of x on the TM and wait (or execute the code of the TM implementation) until T_m receives a subsequent response from the operation. If the response is A_m, the "on event A_m" block is executed. Otherwise, the execution of block B continues. If B is completed successfully (i.e., without any operation returning A_m), T_m sends the TM a commit request, i.e., invokes operation *tryC*(T_m) of the TM. If the response of the request is C_m (or A_m), the "on event C_m" (respectively, "on event A_m") block is executed.

Lemma 7. *An OFTM can be implemented from fo-consensus (and registers).*

Proof (sketch). Implementing an OFTM using fo-consensus (and registers) is a more difficult task. The idea, presented in Algorithm 2, is to use a scheme similar to that underlying DSTM [19], but replace CAS with fo-consensus. Clearly, the transformation is not immediate: fo-consensus is a one-shot object, while a CAS object can change its state infinitely many times. This suggests the need for an unbounded number of fo-consensus objects to implement an OFTM. Basically, the major difference between DSTM and Algorithm 2 is that, because in our algorithm we cannot use CAS, the indirection to object data and to owner transaction's identifier, which are handled in DSTM via single CAS pointers, have to be represented in our OFTM implementation by (infinite) arrays of fo-consensus objects.

[6]The variable k is used here to generate a unique transaction id i, k, where i is the id of process p_i.

Algorithm 2: Implementing an OFTM from fo-consensus and registers

```
  uses: Owner, State – arrays of fo-consensus objects;
        TVar, Aborted, V – arrays of registers (other
        variables are local to transaction T_k)
  initially: Aborted[T_k] = false for every transaction T_k,
             V[x] = ⊥ for every t-variable x, wset = ∅
1  upon read of t-variable x by T_k do
2      return acquire(T_k, x);
3  upon write of value v to t-variable x by T_k do
4      s ← acquire(T_k, x);
5      if s = A_k then return A_k;
6      TVar[x, T_k] ← v;
7      return ok;
8  procedure acquire(T_k, x)
9      if x ∉ wset then
10         version ← 1;
11         state ← initial state of x;
12         v ← V[x];
13         repeat
14             owner ← Owner[x, version].propose(T_k);
15             if owner = ⊥ then return A_k;
16             if owner ≠ T_k then
17                 s ← State[owner].propose(aborted);
18                 if s = ⊥ then return A_k;
19                 if s = committed then
                   state ← TVar[x, owner];
20                 else Aborted[owner] ← true;
21                 if V[x] ≠ v then return A_k;
22             version ← version + 1;
23         until owner = T_k;
24         wset ← wset ∪ {x};
25         TVar[x, T_k] ← state;
26         V[x] ← T_k;
27     else state ← TVar[x, T_k];
28     if Aborted[T_k] then return A_k;
29     return state;
30 upon tryC_k do
31     s ← State[T_k].propose(committed);
32     if s = committed then return C_k;
33     else return A_k;
34 upon tryA_k do
35     return A_k;
```

The idea behind the algorithm is very simple. If a transaction T_k wants to read or update a t-variable x, then T_k must be granted an exclusive, but revocable, ownership on x (procedure *acquire*). To do so, the algorithm first searches for the latest committed state of x (lines 13–23). Then, if there is any live transaction T_i that currently owns object x, T_i is aborted (lines 16–20). Finally, T_k is set as the current owner of x (line 14). Committing or aborting a transaction T_k is done by proposing value `committed`, or `aborted`, to the corresponding fo-consensus $State[T_k]$. Clearly, T_k can commit only if no other transaction aborted T_k before. Also, T_k can be aborted by another transaction T_i only if T_k has not committed yet.

The first time a transaction T_k accesses a t-variable x, T_k creates a new *version* of x. Each version of x is mapped onto a single transaction via the array of fo-consensus objects *Owner*. Transaction T_k creates a new version of x by proposing its id to subsequent elements of $Owner[x, \ldots]$[7] until T_k decides its id (lines 13–23). While doing so, T_k also finds all the transactions that owned x before, i.e., that owned previous versions of x. If any such transaction T_i has committed, T_k reads the latest value written to x by T_i from register $TVar[x, T_i]$ (line 19). If T_i is live, however, i.e., T_i is still the exclusive owner of x, T_k must abort T_i before going further (lines 17–20). This ensures that at any time there is indeed only one owner of x. Once T_k succeeds in becoming an owner of x, T_k saves the newest value of x in register $TVar[x, T_k]$. If transaction T_k accesses x for the second time, T_k is already an owner of x, and so T_k can proceed without going through the array *Owner* again.

For space limitations, we give the proof of correctness of Algorithm 2 in the full version of the paper [15]. □

4.3 Impossibility Result

THEOREM 8. *Fo-consensus cannot implement 3-consensus.*

The intuition behind the proof is the following. We assume, by contradiction, that there exists an algorithm A that implements 3-consensus using only fo-consensus objects and registers. We then derive a contradiction by using a classical "valency argument" [14]. Basically, we show that if A ensures the validity and agreement properties of consensus, then A may violate wait-freedom in some executions, i.e., it may happen that some correct process proposes a value and is never returned a decision value. We do so by proving that any finite low-level history E of A, after which more than one value can be decided, can be extended into a low-level history E' in such a way that still more than one value can be decided after E'. Note that a process p_i may decide value v after a low-level history E only if p_i is sure that no value other than v can be decided by other processes after E (otherwise, agreement could be violated).

PROOF. Assume, by contradiction, that there exists an algorithm A that solves consensus using only fo-consensus objects and registers, in a system of 3 processes: p_1, p_2 and, p_3 (i.e., A implements a 3-consensus object C). Without loss of generality, assume that: (1) the processes can propose only values 0 and 1 to C, (2) every correct process eventually proposes a value to C, and (3) the initial state of the system is fixed.

Every process p_i starts executing A by proposing value 0 or 1 to C. Unless p_i crashes, p_i eventually decides value of 0 or 1. Once any process p_i decides a value v, no other process can decide a value different than v; otherwise, agreement would be violated. Thus, in every infinite low-level history E

[7]Algorithm 2 uses the name (symbol) of a t-variable x to index some of its arrays. This means that, a priori, the algorithm is not dynamic, i.e., it requires that t-variables are allocated statically at the beginning of each execution. Note, however, that the sole purpose of the algorithm is to prove the equivalence result. In fact, its use of unbounded memory and high time complexity make it rather impractical. On the other hand, the algorithm supports an infinite number of t-variables, which makes dynamic allocation of t-variables a non-issue.

of implementation A there is a point after which the decision value is fixed to 0 or 1.

In this proof, we consider only those low-level histories that are *complete*. A history E is complete if it does not contain any *pending* (low-level) operation invocation step. (An invocation of an operation is pending at a process p_i in E, if the invocation is not followed by a (corresponding) response at p_i.) A low-level history E is *valid* if E can be generated by algorithm A. Two histories E and E' are said to be *indistinguishable for a process* p_i, if p_i invokes the same operations and receives the same responses in E as in E'.

An *extension* of E is any low-level history E' of C, such that E is a prefix of E'. We say that E is *0-valent* (respectively, *1-valent*), if in every extension of E only value 0 (respectively, 1) is decided (in C) by any process. A history that is not 0-valent or 1-valent is called *bivalent* [14]. Note that because E defines precisely the state of base objects after E (assuming E is complete), the "valency" of E is also defined.

The result of [14] implies the existence of at least one low-level history of C in which all processes propose a value and that is bivalent. In the following theorem, we prove that, given a bivalent history E, we can find an extension E' of E, $E' \neq E$, such that E' is also bivalent. This means that there exists an infinitely long history that is bivalent. That is, there is a history in which all correct processes propose some values to consensus object C but none of them decides, which violates wait-freedom.

CLAIM 9. *For every finite bivalent complete low-level history E of A there exists a complete valid extension E' of E, $E' \neq E$, such that E' is also bivalent.*

PROOF. By contradiction, assume that there exists a bivalent complete history E, such that every complete extension E' of E is univalent. By [14], for every such history E', every process's next step executed after the last event of E should be an invocation of the *propose* operation on some fo-consensus object.

Denote by $c.propose(p_k, v)$ a sequence of an invocation and a response event of the *propose* operation, executed on fo-consensus object c by process p_k and returning value v. Denote by $[c_r.propose(p_i, v_l), c_s.propose(p_k, v_m)]$ a minimal sequence S of events, such that (1) process p_i invokes the *propose* operation on fo-consensus object c_r and is returned value v_l in S, and (2) process p_k invokes the *propose* operation on fo-consensus object c_s and is returned value v_m in S. Note that the two *propose* operations in S may be concurrent (overlapping), and so one or both of them may abort.

Let v_1, v_2, and v_3 be some values different than $\perp$, for which the following complete extensions of E are valid[8]: $E_1 = E \cdot c_r.propose(p_1, v_1)$, $E_2 = E \cdot c_s.propose(p_2, v_2)$, and $E_3 = E \cdot c_t.propose(p_3, v_3)$. Assume that E_1 and E_3 are 0-valent, and E_2 is 1-valent (the other cases are symmetrical).

First, we show that c_r, c_s, and c_t are the same fo-consensus object. Suppose that c_r and c_s are different objects. But then the valid history $E' = E_1 \cdot c_s.propose(p_2, v_2)$ is indistinguishable for process p_3 from the valid history $E'' = E_2 \cdot c_r.propose(p_1, v_1)$. Thus, if p_1 and p_2 crash just after E' or E'', p_3 will decide the same value after E' and E''—a contradiction with the fact that E' is 0-valent (because E_1 is 0-valent) and E'' is 1-valent (because E_2 is 1-valent). Analogously, we can show that $c_s = c_t$. Hence, $c_r = c_s = c_t = c$.

Consider the following (valid) history, which is a complete extension of history E: $E_4 = E \cdot [c.propose(p_1, \perp), c.propose(p_3, \perp)]$. There are two cases to consider:

Case 1: E_4 is 0-valent. History E_4 is indistinguishable for p_2 from history E, and fo-consensus c is in the same state after E and E_4. Hence, the extension $E' = E_4 \cdot c.propose(p_2, v_2)$ of E_4 is valid and indistinguishable for process p_2 from history E_2. But E_2 is 1-valent, and so in every extension of E' process p_2 will decide 1 if p_1 and p_3 crash just after E_4—a contradiction with the fact that E' is 0-valent (because E_4 is 0-valent).

Case 2: E_4 is 1-valent. Consider the following (valid) history: $E_5 = E \cdot [c.propose(p_1, \perp), c.propose(p_2, \perp)]$. History E_5 is indistinguishable for process p_1 from history E_4, and the state of fo-consensus c is the same after E_4 and E_5. Hence, E_5 is 1-valent: otherwise, if p_2 and p_3 crashed just after E_4 or E_5, p_1 could not decide different values after E_4 (which is 1-valent) and after E_5.

History E_5 is indistinguishable for process p_3 from history E, and fo-consensus c is in the same state after E and E_5. Hence, the extension $E' = E_5 \cdot c.propose(p_3, v_3)$ of E_5 is valid and indistinguishable for process p_3 from history E_3. But E_3 is 0-valent, and so in every extension of E' process p_3 will decide 0 if p_1 and p_2 crash just after E_5—a contradiction with the fact that E' is 1-valent (because E_5 is 1-valent). □

From Lemma 6, Lemma 7, Theorem 8, and the claim of [6] that consensus can be implemented from fo-consensus and registers in a system of 2 processes, we have:

COROLLARY 10. *The consensus number of an OFTM equals 2.*

5. IMPOSSIBILITY OF STRICT DISJOINT-ACCESS-PARALLELISM

In this section, we prove that no OFTM can be strictly disjoint-access-parallel. We first define precisely our notion of strict disjoint-access-parallelism. Then, we prove our result. We discuss its scope in Section 6.

5.1 Definitions

To define the notion of strict disjoint-access-parallelism, we distinguish base object operations that modify the state of the 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, an STM is *strictly disjoint-access-parallel* if it ensures that processes executing transactions which access disjoint sets of t-variables do not conflict on common base objects. More precisely:

DEFINITION 11. *We say that an STM implementation I is* strictly disjoint-access-parallel *if, for every low-level history E of I and every two transactions T_i and T_k, if T_i and T_k conflict on a base object, then T_i and T_k both access some common t-variable.*

[8]We denote by $E \cdot S$ the concatenation of history E and sequence S of events.

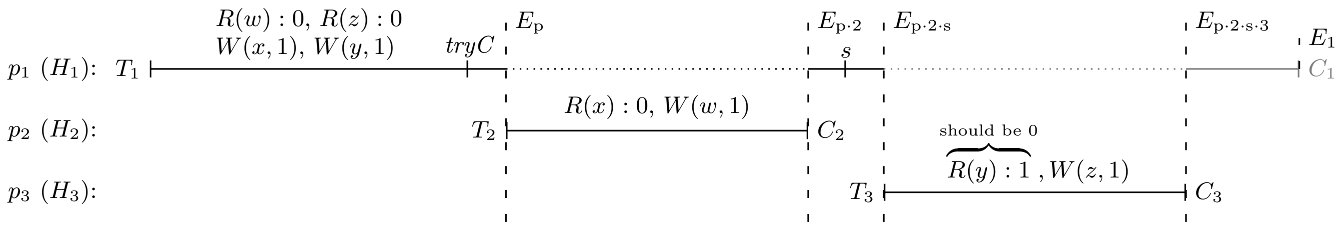

Figure 2: Execution used in the strict disjoint-access-parallelism impossibility proof. $R(x) : 0$ denotes a read of a t-variable x returning value 0, and $W(x, 1)$ denotes a write of value 1 to a t-variable x.

5.2 Impossibility Result

THEOREM 12. *No OFTM is strictly disjoint-access-parallel.*

The intuition behind the proof of the result is the following (the full proof is in the full version of the paper [15]). We assume, by contradiction, that there is an OFTM that is strictly disjoint-access-parallel, and we consider the scenario depicted in Figure 2, with transactions T_1, T_2, and T_3 involved in low-level histories E_1 and $E_{\mathrm{p}\cdot 2\cdot \mathrm{s}\cdot 3}$. The transactions access t-variables x, y, w, and z, initialized to 0. Transaction T_1 reads value 0 from w and z, and writes value 1 to both x and y, while transactions T_2 and T_3 read, respectively, x and y, and write value 1 to, respectively, w and z. In low-level history E_1, transaction T_1 executes alone. Thus, T_1 modifies x and y and eventually commits (by the properties of an OFTM, T_1 cannot be forcefully aborted in E_1).

Suppose now that process p_1, which executes T_1, gets suspended at some point t in E_1 and either T_2 or T_3 is executed and committed before p_1 resumes taking steps. (Note that p_2 and p_3 cannot wait for p_1 to take steps, because the system is asynchronous and p_1 might have crashed; neither T_2 nor T_3 can be forcefully aborted, because p_1 does not take any steps when any of these transactions are executed.) Clearly, if t is before the invocation of $tryC(T_1)$, then T_2 and T_3 cannot read value 1 from x or y. This is because T_1 might invoke $tryA(T_1)$ instead of $tryC(T_1)$, in which case value 1 may never be seen by any committed transaction. If t is after the commit event of T_1, then both T_2 and T_3 can only read value 1 from x or y—otherwise serializability would be violated, because T_1 reads value 0 from w and z. This means that there must be some "critical" step s, such that (1) if t is before s, then neither T_2 nor T_3 can read 1 from x or y, and (2) if t is after s then at least one of the two transactions, say T_3, reads 1 from x or y (the other case is symmetrical).

Consider a low-level history $E_{\mathrm{p}\cdot 2\cdot \mathrm{s}\cdot 3}$ in which transaction T_2 is executed and committed before step s, then p_1 executes step s, and finally transaction T_3 is executed and committed (with p_1 being suspended during the execution of T_2 and T_3). By our assumption, T_2 reads 0 from x in $E_{\mathrm{p}\cdot 2\cdot \mathrm{s}\cdot 3}$. This means that T_1 cannot commit, as the conflict between T_1 and T_2 is not resolvable without aborting one of the two transactions or violating serializability. Transaction T_3 executes after step s and, as T_2 and T_3 access different t-variables, process p_3 cannot read any base objects that are modified by p_2. Hence, transaction T_2 is effectively "invisible" to p_3. But then T_3 reads value 1 from y. However, this means that T_1, which is the only transaction that writes to y, must be committed—otherwise serializability is violated. Hence, on the one hand, T_1 must commit, but, on the other hand, T_1 cannot commit, and so we reach a contradiction.

6. SCOPING THE RESULTS

In this section, we discuss the scope of our results.

Obstruction-freedom. The results in Sections 4 (equivalence to fo-consensus) and 5 (impossibility of strict disjoint-access-parallelism) are proved for OFTMs. It is worth discussing, whereas those results hold also for weaker definitions that are presented, and compared, in Section 3.

Theorems 4 and 5 imply, together with Lemmas 6 and 7, that an ic-OFTM and an eventual ic-OFTM are also equivalent to fo-consensus, and thus have consensus number of 2. Theorem 4 also implies, together with Theorem 12, that an ic-OFTM cannot be strictly disjoint-access-parallel.

However, it is not obvious that strict disjoint-access-parallelism is impossible for an eventual ic-OFTM. To prove that, we go back to the proof of Theorem 12. In the proof, transactions T_2 and T_3 could not be forcefully aborted. However, an eventual ic-OFTM could abort T_2 and T_3, because T_1 is concurrent to both T_2 and T_3. But process p_1 does not take any steps while T_2 and T_3 execute. Hence, p_2 and p_3 cannot say whether p_1 has crashed or is just suspended (as the system is asynchronous). Therefore, if we keep restarting transactions T_2 and T_3 (i.e., their computations), those transactions will eventually commit. Hence, we can reach the same contradiction as in the proof of Theorem 12: even eventual ic-OFTMs cannot be strictly disjoint-access-parallel.

Opacity. Serializability is a relatively weak safety property for a TM. Most STM implementations ensure a stronger correctness criterion called *opacity* [16], which adds to serializability the requirements that (1) all transactions (even non-committed ones) always observe a consistent state of the system, and (2) the real-time order of transactions is preserved. An OFTM that ensures opacity is still equivalent to fo-consensus—Algorithm 2, in fact, guarantees opacity (see its correctness proof in [15]). Hence, an OFTM ensuring opacity has still consensus number 2, i.e., opacity does not make an OFTM able to implement 3-consensus. Also, the impossibility of strict disjoint-access-parallelism clearly holds for any OFTM that ensures opacity.

Arbitrary t-variables. In the proofs of the results presented in this paper, we considered only t-variables that can be read and written (i.e., transactional registers). Some of the results may not hold if read-write t-variables are not provided by an OFTM. For example, an OFTM that supports only write-only t-variables (i.e., where transactions cannot read transactional data) can be trivially implemented without any base objects, and thus has a consensus number of 1. However, read-write t-variables are considered essential, and so they are provided by every existing TM.

It is interesting, however, to see what happens when an OFTM supports t-variables that export some operations in addition to *read* and *write*. Clearly, such an OFTM is strictly more difficult to implement than an OFTM that supports only registers. Hence, it cannot be strictly disjoint-access-parallel, and cannot have consensus number lower than 2.

Now, consider an OFTM implementation A that supports only read-write t-variables, and let Q be a type (class) of an object that exports operations other than *read* and *write*. Let B be an implementation of an object of type Q, in a sequential, non-transactional system, that uses only read-write variables. Using a single instance of A, we can implement an OFTM that provides t-variables of type Q. Basically, whenever a transaction invokes an operation *op* of a t-variable of type Q, we follow the implementation B, using read-write t-variables instead of non-transactional variables. Because all operations performed by a transaction should appear as if they were executed atomically, B executed by a transaction must provide a correct implementation of an object of type Q. This means that supporting t-variables that export operations other than *read* and *write* does not increase the computational power of an OFTM, i.e., its consensus number[9].

Disjoint-access-parallelism. The original notion of disjoint-access-parallelism, introduced in [23], allows for transactions that are *indirectly* connected via other transactions to conflict on common base objects. For example, if a transaction T_1 accesses t-variable 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-variables. Disjoint-access-parallelism allows then the processes executing T_1 and T_2 to delay one another. Disjoint-access-parallelism in the sense of [23] can be ensured by an OFTM implementation, e.g., DSTM.

7. CONCLUDING REMARKS

Obstruction-freedom. The concept of obstruction-free shared object implementations has been first informally introduced in [20]. A formalization of the concept was then proposed in [6]. In short, the definition of [6] requires operations to return if there is no *step contention*. If there is, the operations could abort but need to return control to the application, i.e., rather than livelock forever. An alternative definition, based on interval contention, was proposed in [4] through the concept of "abortable" objects. In particular, it is argued there that a definition based on step contention (as in [6]) is not composable.

The concept of obstruction-free TM implementation was first informally discussed in [19]. Many OFTMs have been proposed since then, including DSTM [19], ASTM [26], RSTM [1] and NZTM [30]. However, until our paper, there has been no formal definition of the concept. Our definition of an OFTM is a logical extension of that in [6] to transactions. However, we also consider (in Section 3) alternative definitions (e.g., inspired by [4]) and discuss their computational equivalence to our definition. We point out the fact that our results apply also to these alternative definitions.

Limitations of OFTMs. The first paper to discuss the limitations of OFTMs was [12]. The paper argues about several practical disadvantages of ensuring obstruction-freedom, and discusses how those can be overcome using simple, lock-based schemes. In particular, the paper points out the necessity for an OFTM to use indirection (a claim questioned by [30]), which results in cache-locality problems, and the difficulty of limiting the number of concurrent transactions to the number of physical processors. Our consensus impossibility result is clearly of different nature than the claims in [12]. The impossibility of strict disjoint-access-parallelism is indeed related to cache issues. However, those issues result from transactional metadata accessed by transactions that are not directly related, rather than from indirections towards states of transactional objects [12].

It is worth noting that some lower bounds on obstruction-free implementations have already been established. In [5], space and time complexity lower bounds for obstruction-free implementations of so-called *perturbable objects* have been derived. As an OFTM can be used to implement any perturbable object, these lower bounds naturally hold also for OFTMs. However, the lower bounds concerning time and space complexity are clearly of a different nature than our consensus number proof and our strict disjoint-access-parallelism impossibility. The last result in [5], which is a lower bound on the number of stalls a process may incur in some executions, is similar in scope to our strict disjoint-access-parallelism proof. However, this particular result of [5] holds only when there are no aborts, which is clearly not the case for OFTMs. In [16], a complexity lower bound for a class of STM implementations that ensure opacity is proved. However, the bound is not inherent to OFTMs: it holds for OFTMs as well as for lock-based STMs.

Consensus number of OFTMs. In [6], a "fail-only" consensus object is introduced and shown to have consensus number at least 2. We use this object as an intermediate abstraction for our first result: that is, we (1) prove than an OFTM is equivalent to a "fail-only" consensus, and (2) show that a "fail-only" consensus (and thus an OFTM) has consensus number *at most 2*. The proof of (2) uses the classical "valency argument" first introduced in [14].

It is also important to notice that the consensus number of objects roughly similar to TMs have already been determined. In particular, in [2, 28] upper and lower bounds on the consensus number of several classes of *multi-objects* are given. Multi-objects, however, differ from TMs in that: (1) the sequence of operations that are to be executed atomically (a multi-object operation) is known in advance (unlike in transactions), (2) a multi-object operation cannot abort, and (3) a multi-object consists of a set of objects with the same type and a specified, finite consensus number (transactions can use objects of any type and in any way).

8. ACKNOWLEDGEMENTS

We would like to thank Hagit Attiya, Petr Kouznetsov, Eshcar Hillel, and the anonymous reviewers for their help and valuable comments.

[9]However, an OFTM that supports t-variables of type Q directly may be, in principle, more efficient than an OFTM that implements such t-variables using transactional registers. For example, commutativity or conflict relations between some operations of Q may be exploited to allow for more concurrency between transactions.

9. REFERENCES

[1] RSTM—the Rochester software transactional memory runtime. http://www.cs.rochester.edu/research/synchronization/rstm.

[2] Y. Afek, M. Merritt, and G. Taubenfeld. The power of multi-objects. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[3] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations (extended abstract). In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 111–120. ACM, 1997.

[4] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, New York, NY, USA, 2007. ACM.

[5] H. Attiya, R. Guerraoui, D. Hendler, and P. Kouznetsov. Synchronizing without locks is inherently expensive. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2006.

[6] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, 2005.

[7] H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006.

[8] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

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

[11] D. Dice and N. Shavit. What really makes transactions fast? In *Proceedings of the 1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.

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

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

[14] 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, April 1985.

[15] R. Guerraoui and M. Kapałka. On obstruction-free transactions. Technical report, EPFL, 2008.

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

[17] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

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

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[20] M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

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

[22] 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.

[23] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

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

[25] J. R. Larus and R. Rajwar. *Transactional Memory.* Morgan&Claypool, 2007.

[26] V. J. Maranthe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 354–368, 2005.

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

[28] E. Ruppert. Consensus numbers of multi-objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

[29] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213. August 1995.

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